Practical Parallel Divide-and-Conquer Algorithms

Jonathan C. Hardwick

December 1997

CMU-CS-97-197

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Thesis Committee:
Guy Blelloch, Chair
Adam Beguelin
Bruce Maggs
Dennis Gannon, Indiana University

19980415 014

Carnegie Mellon

School of Computer Science

DOCTORAL THESIS

in the field of

COMPUTER SCIENCE

Practical Parallel Divide-and-Conquer Algorithms
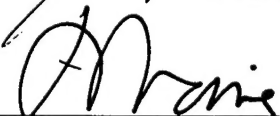
JONATHAN HARDWICK

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:
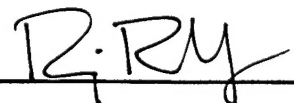
_____   12/17/97
THESIS COMMITTEE CHAIR   DATE

_____   12-18-97
DEPARTMENT HEAD   DATE

APPROVED:

_____   12-18-97
DEAN   DATE

For Barbara and Richard

ii

Abstract

Nested data parallelism has been shown to be an important feature of parallel languages, allowing the concise expression of algorithms that operate on irregular data structures such as graphs and sparse matrices. However, previous nested data-parallel languages have relied on a vector PRAM implementation layer that cannot be efficiently mapped to MPPs with high inter-processor latency. This thesis shows that by restricting the problem set to that of data-parallel divide-and-conquer algorithms I can maintain the expressibility of full nested data-parallel languages while achieving good efficiency on current distributed-memory machines.

Specifically, I define the team parallel model, which has four main features: data-parallel operations within teams of processors, the subdivision of these teams to match the recursion of a divide-and-conquer algorithm, efficient single-processor code to exploit existing serial compiler technology, and an active load-balancing system to cope with irregular algorithms. I also describe Machiavelli, a toolkit for parallel divide-and-conquer algorithms that implements the team parallel model. Machiavelli consists of simple parallel extensions to C, and is based around a distributed vector datatype. A preprocessor generates both serial and parallel versions of the code, using MPI as its parallel communication mechanism to assure portability across machines. Load balancing is performed by shipping function calls between processors.

Using a range of algorithm kernels (including sorting, finding the convex hull of a set of points, computing a graph separator, and matrix multiplication), I demonstrate optimization techniques for the implementation of divide-and-conquer algorithms. An important feature of team parallelism is its ability to use efficient serial algorithms supplied by the user as the base case of recursion. I show that this allows parallel algorithms to achieve good speedups over efficient serial code, and to solve problems much larger than those which can be handled on one processor. As an extended example, a Delaunay triangulation algorithm implemented using the team-parallel model is three times as efficient as previous parallel algorithms.

Acknowledgements

Contents

List of Figures

x

List of Tables

Chapter 1

Introduction

I like big machines.
—Fred Rogers, *Mister Rogers' Neighborhood*

It's hard to program parallel computers. Dealing with many processors at the same time, either explicitly or implicitly, makes parallel programs harder to design, analyze, build, and evaluate than their serial counterparts. However, using a fast serial computer to avoid the problems of parallelism is often not enough. There are always problems that are too big, too complex, or whose results are needed too soon.

Ideally, we would like to program parallel computers in a model or language that provides the same advantages that we have enjoyed for many years in serial languages: portability, efficiency, and ease of expression. However, it is typically impractical to extract parallelism from sequential languages. In addition, previous parallel languages have generally ignored the issue of *nested parallelism*, where the programmer exposes multiple sources of parallelism in an algorithm. Supporting nested parallelism is particular important for *irregular* algorithms, which operate on non-uniform data structures (for example, sparse arrays).

My thesis is that, by targetting a particular class of algorithms, namely divide-and-conquer algorithms, we can achieve our goals of efficiency, portability, and ease of expression, even for irregular algorithms with nested parallelism. In particular, I claim that:

- Divide-and-conquer algorithms can be used to express many computationally significant problems, and irregular divide-and-conquer algorithms are an important subset of this class. We would like to be able to implement these algorithms on parallel computers in order to solve larger and harder problems.

- There are natural mappings of these divide-and-conquer algorithms onto the processors of a parallel machine such that we can support nested parallelism. In this dissertation I

1

use recursive subdivision of processor teams to match the dynamic run-time behavior of the algorithms.

- A programming system that incorporates these mappings can be used to generate portable and efficient programs using message-passing primitives. The use of asynchronous processor teams reduces communication requirements, and enables efficient serial code to be used at the leaves of the recursion tree.

The rest of this chapter is arranged as follows. In Section 1.1 I discuss previous serial and parallel languages. In Section 1.2 I discuss the uses and definition of divide-and-conquer algorithms, and outline the *team parallel* model, which provides a general mapping of these algorithms onto parallel machines. In Section 1.3 I describe the Machiavelli system, which is my implementation of the team parallel model for distributed-memory machines, and give an overview of performance results. In Section 1.5 I define the assumptions and limits of the team-parallel model and the Machiavelli implementation. In Section 1.6 I list the contributions of this work. Finally, Section 1.7 describes the organization of the dissertation.

1.1   Previous Work

As explained above, ideally we could exploit parallelism using existing serial models and languages. The serial world has many attractions. There has been a single dominant high-level abstraction (or model) of serial computing, the RAM (Random Access Machine). Historically, this has corresponded closely to the underlying hardware of a von Neumann architecture. As a result, it has been possible to design and analyze algorithms independent of the language in which they will be written and the platform on which they will run. Over time, we have developed widely-accepted high-level languages that can be compiled into efficient object code for virtually any serial processor. We would like to retain these virtues of ease of programming, portability, and efficiency, while running the code on parallel machines.

Unfortunately, automatically exploiting parallelism in a serial program is arguably even harder than programming a parallel computer from scratch. Run-time instruction-level parallelism in a processor—based on multiple functional units, multi-threading, pipelining, and similar techniques—can extract at most a few fold speedup out of serial code. Compile-time approaches can do considerably better in some cases, using a compiler to spot parallelism in serial code and replace it with a parallel construct, and much work has been done at the level of loop bodies (for a useful summary, see [Ghu97]). However, a straightforward parallel translation of a serial algorithm to solve a particular problem is often less efficient—in both theoretical and practical terms—than a program that uses a different algorithm more suited to

2

parallel machines. An optimal parallelizing compiler should therefore recognize any serial formulation of such an algorithm and replace it with the more efficient parallel formulation. This is impractical given the current state of automated code analysis techniques.

1.1.1 Parallel languages and models

An alternative approach is to create a new parallel model or language that meets the same goals of ease of programming, portability and general efficiency. Unfortunately, the last two goals are made more difficult by the wide range of available parallel architectures. Despite shifts in market share and the demise of some manufacturers, users can still choose between tightly-coupled shared-memory multiprocessors such as the SGI Power Challenge [SGI94], more loosely coupled distributed-memory multicomputers such as the IBM SP2 [AMM+95], massively-parallel SIMD machines such as the MasPar MP-2, vector supercomputers such as the Cray C90 [Oed92], and loosely coupled clusters of workstations such as the DEC Super-Cluster. Network topologies are equally diverse, including 2D and 3D meshes on the Intel Paragon [Int91] and ASCI Red machine, 3D tori on the Cray T3D [Ada93] and T3E [Sco96], butterfly networks on the IBM SP2, fat trees on the Meiko CS-2, and hypercube networks on the SGI Origin2000 [SGI96]. With extra design axes to specify, parallel computers show a much wider range of design choices than do serial machines, with each choosing a different set of tradeoffs in terms of cost, peak processor performance, memory bandwidth, interconnection technology and topology, and programming software.

This tremendous range of parallel architectures has spawned a similar variety of theoretical computational models. Most of these are variants of the original CRCW PRAM model (Concurrent-Read Concurrent-Write Parallel Random Access Machine), and are based on the observation that although the CRCW PRAM is probably the most popular theoretical model amongst parallel algorithm designers, it is also the least likely to ever be efficiently implemented on a real parallel machine. That is, it is easily and efficiently portable to *no* parallel machines, since it places more demands on the memory system in terms of access costs and capabilities than can be economically supplied by current hardware. The variants handicap the ideal PRAM to resemble a more realistic parallel machine, resulting in the locality-preserving H-PRAM [HR92a], and various asynchronous, exclusive access, and queued PRAMs. However, none of these models have been widely accepted or implemented.

Parallel models which proceed from machine characteristics and then abstract away details—that is, "bottom-up" designs rather than "top-down"—have been considerably more successful, but tend to be specialized to a particular architectural style. For example, LogP [CKP+93] is a low-level model for message-passing machines, while BSP [Val90] defines a somewhat higher-level model in terms of alternating phases of asynchronous computation and synchronizing communication between processors. Both of these models try to accurately characterize

3

the performance of any message-passing network using just a few parameters, in order to allow a programmer to reason about and predict the behavior of their programs.

However, the two most successful recent ways of expressing parallel programs have been those which are arguably not models at all, being defined purely in terms of a particular language or library, with no higher-level abstractions. Both High Performance Fortran [HPF93] and the Message Passing Interface [For94] have been created by committees and specified as standards with substantial input from industry, which has helped their widespread adoption. HPF is a full language that extends sequential Fortran with predefined parallel operations and parallel array layout directives. It is typically used for computationally intensive algorithms that can be expressed in terms of dense arrays. By contrast, MPI is defined only as a library to be used in conjunction with an existing sequential language. It provides a standard message-passing model, and is a superset of previous commercial products and research projects such as PVM [Sun90] and NX [Pie94]. Note that MPI is programmed in a control-parallel style, expressing parallelism through multiple paths of control, whereas HPF uses a data-parallel style, calling parallel operations from a single thread of control.

1.1.2   Nested and irregular parallelism

Neither HPF or MPI provide direct support for nested parallelism or irregular algorithms. For example, consider the quicksort algorithm in Figure 1.1. Quicksort [Hoa62] will be used as an example of an irregular divide-and-conquer algorithm throughout this dissertation. It was chosen because it is a well-known and very simple divide-and-conquer algorithm that nevertheless illustrates many of the implementation problems faced by more complex algorithms. The irregularity comes from the fact that the two subproblems that quicksort creates are typically not of the same size; that is, the divide-and-conquer algorithm is *unbalanced*.

Although it was originally written to describe a serial algorithm, the pseudocode in Figure 1.1 contains both data-parallel and control-parallel operations. Comparing the elements of the sequence S to the pivot element a, and selecting the elements for the new subsequences S_1, S_2, and S_3 are inherently data-parallel operations. Meanwhile, recursing on S_1 and S_3 can be implemented as a control-parallel operation by performing two recursive calls in parallel on two different processors.

Note that a simple data-parallel quicksort (such as one written in HPF) cannot exploit the control parallelism that is available in this algorithm, while a simple control-parallel quicksort (such as one written in a sequential language and MPI) cannot exploit the data parallelism that is available. For example, a simple control-parallel divide-and-conquer implementation would initially put the entire problem onto a single processor, leaving the rest of the processors unused. At the first divide step, one of the subproblems would be passed to another processor.

4

```
procedure QUICKSORT(S):
if S contains at most one element
then
  return S
else
  begin
    choose an element a randomly from S;
    let S₁, S₂ and S₃ be the sequences of elements in S
      less than, equal to, and greater than a, respectively;
    return (QUICKSORT(S₁) followed by S₂ followed by QUICKSORT(S₃))
  end
```

Figure 1.1: Quicksort, a simple irregular divide-and-conquer algorithm. Taken from [AHU74].

At the second divide step, a total of four processors would be involved, and so on. The parallelism achieved by this algorithm is proportional to the number of threads of control, which is greatest at the end of the algorithm. By contrast, a data-parallel divide-and-conquer quicksort would serialize the recursive applications of the function, executing one at a time over all of the processors. The parallelism achieved by this algorithm is proportional to the size of the sub-problem being operated on at any instant, which is greatest at the beginning of the algorithm. Towards the end of the algorithm there will be fewer data elements in a particular function application than there are processors, and so some processors will remain idle.
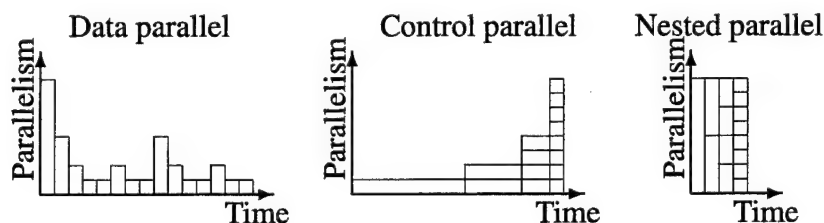
Figure 1.2: Available parallelism versus time of a divide-and-conquer algorithm expressed in three parallel language models.

By simultaneously exposing both nested sources of parallelism, a *nested parallel* implementation of quicksort can achieve parallelism proportional to the total data size throughout the algorithm, rather than only achieving full parallelism at either the beginning (in data parallelism) or the end (in control parallelism) of the algorithm. This is shown in Figure 1.2 for a regular (balanced) divide-and-conquer algorithm. To handle unbalanced algorithms, an implementation must also be able to cope with subtasks of uneven sizes. As an example, Figure 1.3

5

shows the nested recursion in a divide-and-conquer algorithm to implement Delaunay triangulation. Note the use of an unbalanced convex hull algorithm within the balanced triangulation algorithm. Chapter 2 discusses previous parallel models and languages and their support for nested parallelism, while Chapter 7 presents the Delaunay triangulation algorithm.
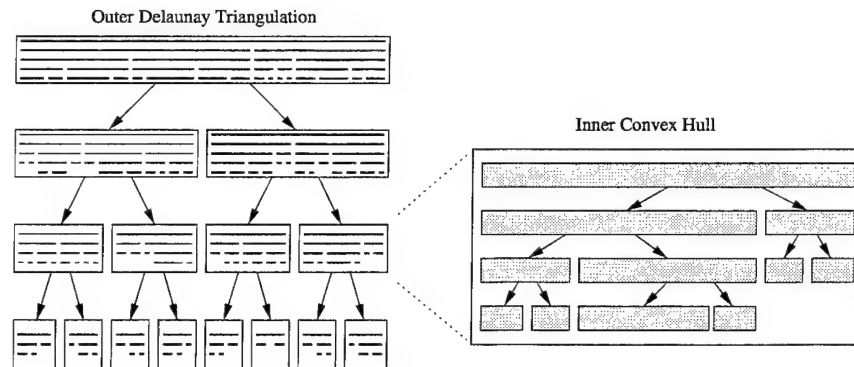
Figure 1.3: Representation of nested recursion in Delaunay triangulation algorithm by Blelloch, Miller and Talmor [BMT96]. Each recursive step of the balanced outer divide-and-conquer triangulation algorithm uses as a substep an inner convex hull algorithm that is also divide-and-conquer in style, but is not balanced.

1.2 Divide-and-conquer Algorithms and Team Parallelism

Divide-and-conquer algorithms solve a problem by splitting it into smaller, easier-to-solve parts, solving the subproblems, and then combining the results of the subproblems into a result for the overall problem. The subproblems typically have the same nature as the overall problem (for example, sorting a list of numbers), and hence can be solved by a recursive application of the same algorithm. A base case is needed to terminate the recursion. Note that a divide-and-conquer algorithm is inherently dynamic, in that we do not know all of the subtasks in advance.

Divide-and-conquer has been taught and studied extensively as a programming paradigm, and can be found in any algorithm textbook (e.g., [AHU74, Sed83]). In addition, many of the most efficient and widely used computer algorithms are divide-and-conquer in nature. Examples from various fields of computer science include algorithms for sorting, such as merge-sort and quicksort, for computational geometry problems such as convex hull and closest pair, for graph theory problems such as traveling salesman and separators for VLSI layout, and for numerical problems such as matrix multiplication and fast Fourier transforms.

The subproblems that are generated by a divide-and-conquer algorithm can typically be solved independently. This independence allows the subproblems to be solved simultaneously,

6

and hence divide-and-conquer algorithms have long been recognized as possessing a potential source of control parallelism. Additionally, all of the previously described algorithms can be implemented in terms of data-parallel operations over *collection-oriented* data types such as sets or sequences [SB91], and hence we can also exploit data parallelism in their implementation. However, we must still define a model in which to express this parallelism. Previous models have included fork-join parallelism, and the use of processor groups, but both of these models have severe limitations. In the fork-join model available parallelism and the maximum problem size is greatly limited, while group-parallel languages have been limited in their portability, performance, and/or ability to handle irregular divide-and-conquer algorithms (previous work will be further discussed in Chapter 2). To overcome these problems, I define the *team parallel* model.

1.2.1   The team parallel model

The team parallel model uses data parallelism within teams of processors acting in a control-parallel manner. The basic data structures are distributed across the processors of a team, and are accessible via data-parallel primitives. The divide stage of a divide-and-conquer algorithm is then implemented by dividing the current team of processors into two subteams, distributing the relevant data structures between them, and then recursing independently within each subteam. This natural mixing of data parallelism and control parallelism allows the team parallel model to easily handle nested parallelism.

There are three important additions to this basic approach. First, in the case when the algorithm is unbalanced, the subteams of processors may not be equally sized—that is, the division may not always be into equal pieces. In this case we choose the sizes of the subteams in relation to the cost of their subtasks. However, this passive load-balancing method is often not enough, because the number of processors is typically much smaller than the problem size, and hence granularity effects mean that some processors will end up with an unfair share of the problem when the teams have recursed down to the point where each consists of a single processor. Thus, the second important addition is an active load-balancing system. In this dissertation I use function-shipping to farm out computation to idle processors. This can be seen as a form of remote procedure call [Nel81]. Finally, when the team consists of a single processor that processor can use efficient sequential code, either by calling a version of the parallel algorithm compiled for a single processor or by using a completely different sequential algorithm. Chapter 3 provides additional analysis of divide-and-conquer models, and fully defines the team parallel model.

7

1.3 Machiavelli, an Implementation of Team Parallelism

Machiavelli is a particular implementation of the team parallel model. It is presented as an extension to the C programming language. The basic data-parallel data structure is a *vector*, which is distributed across all the processors of the current team. Vectors can be formed from any of the basic C datatypes, and from any user-defined datatypes. Machiavelli supplies a variety of basic parallel operations that can be applied to vectors (scans, reductions, permutations, appends, etc), and in addition allows the user to construct simple data-parallel operations of their own. A special syntax is used to denote recursion in a divide-and-conquer algorithm.

Machiavelli is implemented using a simple preprocessor that translates the language extensions into C plus calls to MPI. The use of MPI ensures the greatest possible portability across both distributed-memory machines and shared-memory machines. Data-parallel operations in Machiavelli are translated into loops over sections of a vector local to each processor, while the predefined basic parallel operations are mapped to MPI functions. The recursion syntax is translated into code that computes team sizes, subdivides the processors, redistributes the argument vectors to the appropriate subteams, and then recurses in a smaller team. User-defined datatypes are automatically mapped into MPI datatypes, allowing them to be used in any Machiavelli operation (for example, sending a vector of user-defined point structures instead of sending two vectors of x and y coordinates). This also allows function arguments and results to be transmitted between processors, allowing for the RPC-like active load-balancing system described in Section 1.2. The importance of the load-balancing system can be seen in Figure 1.4, which shows quicksort implemented with and without load balancing. Additionally, Machiavelli supports both serial compilation of parallel functions (removing the MPI constructs), and the overriding of these default functions with user-defined serial functions that can implement a more efficient algorithm. Machiavelli is described further in Chapter 4.

1.4 Summary of Results

I have evaluated the performance of Machiavelli in three ways: by benchmarking individual primitives, by benchmarking small algorithmic kernels, and by benchmarking a major application. To summarize the results, I will use *parallel efficiency*, which is efficiency relative to the notionally perfect scaling of a good serial algorithm. For example, a parallel algorithm that runs twice as fast on four processors as the serial version does on one processor has a parallel efficiency of 50%. I will also use the extreme case of fixing the problem size as the largest which can be solved on one processor. For large numbers of processors, this increases the effects of parallel overheads, since there is less data per processor. However, it allows a direct comparison to single-processor performance.

8
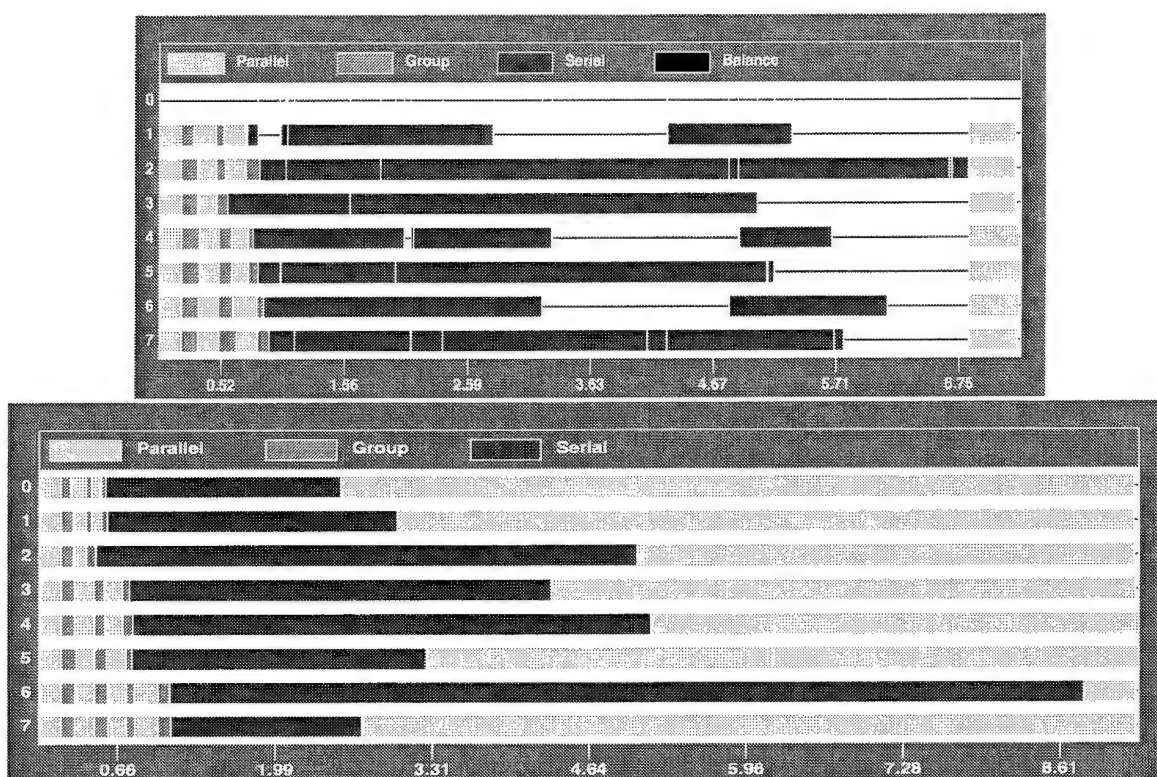
Figure 1.4: Behavior of eight processors on an IBM SP2 running parallel quicksort with (top) and without (bottom) load balancing, shown on the same time scale. Note that one processor acts as a dedicated manager when load balancing.

Machiavelli primitives are typically implemented in terms of local per-processor operations combined with a handful of MPI operations—for example, a local sum followed by an MPI sum across processors. This generally results in predictable performance. Parallel efficiency for primitives that do not use any MPI operations, such as the distribution of a scalar across a vector, is near 100% (or better, due to caching effects). Where MPI operations are used the parallel efficiency can be much lower, reflecting the cost of communication between processors. For example, for an eight-processor SGI Power Challenge the measured parallel efficiencies range from 11% (for a fetch operation involving all-to-all communication) to 99% (for a reduction operation). These primitive benchmarks are described further in Chapter 5.

I have also tested four different divide-and-conquer algorithmic kernels: quicksort, a convex hull algorithm, a geometric separator, and a dense matrix multiplication algorithm. The parallel efficiency of these kernels ranges between 22% and 26% on a 32-processor SP2, between 27% and 41% on a 32-processor T3D, and between 41% and 58% on a 16-processor SGI Power Challenge. These algorithmic benchmarks are described further in Chapter 6.

9
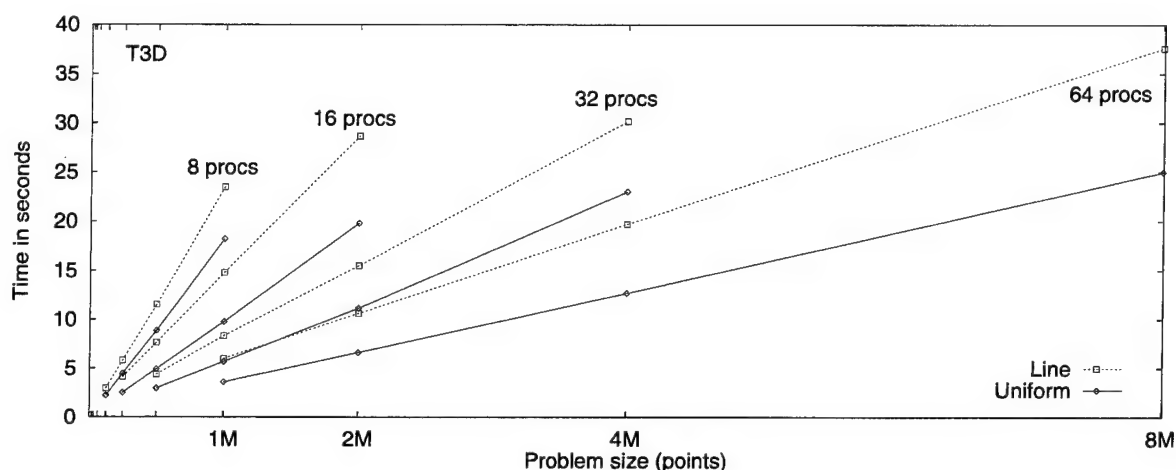
Figure 1.5: Performance of a Machiavelli implementation of Delaunay triangulation on the Cray T3D, showing the time to triangulate 16k-128k points per processor for a range of machine sizes, and for uniform and non-uniform ("line") distributions.

Finally, I developed a major application, Delaunay triangulation, using the Machiavelli primitives. This implements a new divide-and-conquer algorithm by Blelloch, Miller and Talmor [BMT96], and is designed to overcome two problems of previous algorithms: slowdowns of up to an order of magnitude on irregular input point sets, and poor parallel efficiency. The implementation is at most 50% slower on non-uniform datasets than on uniform datasets, and also has good absolute performance, with a parallel efficiency of greater than 50% on machine sizes and problem sizes of interest. Figure 1.5 illustrates the performance of this application on the Cray T3D, showing good scalability from 8 to 64 processors, and good performance on both uniform and non-uniform input distributions. The Delaunay application is described further in Chapter 7.

1.5   Limits of the Dissertation

There are several aspects of the research area that I do not attempt to cover in this dissertation. In addition, there are several explicit assumptions that are made about the current state of parallel computing, and that could invalidate the thesis if they were to change.

1.5.1   What the dissertation does not cover

The Machiavelli system includes a simple preprocessor, which has roughly the same semantic power as C++'s template system. I have not provided a full compiler for the language, nor have

10

I provided a compiler into Machiavelli from an existing language such as NESL [BHS+94].

I have not investigated optimizations specific to balanced divide-and-conquer algorithms. These represent a smaller range of computational problems, and their parallel behavior and implementation has been extensively covered in previous work, as we will see in Chapter 2.

I have not provided support for *impure* divide-and-conquer algorithms, in which function calls are not independent (that is, they can exchange data with each other during computation). This is a restriction by the use of MPI, rather than an inherent limitation of the team parallel model. I discuss ways to remove this restriction in Chapter 8.

I have not specialized my implementation for a particular architecture or machine, since this would work against the goal of demonstrating portability. I therefore avoid relying on techniques such as multithreading and shared memory. Similarly, I restrict myself to using software packages for which there are widely-available implementations (namely MPI), rather than relying on vendor-specific packages.

I have not provided a complete language. For example, there are no parallel input/output operations—all test data for benchmarking is generated at run time. Good models and implementations of parallel I/O remain an active area of research, and a separate topic in their own right. However, I speculate on integrating I/O into the team parallel model in Chapter 8.

1.5.2   What assumptions the thesis makes

I assume that the divide-and-conquer algorithm being implemented has data parallelism available in its operations—that is, the division, merging, and solution steps can be expressed in terms of operations over collections of data. This is generally true, since by expressing the algorithm in a divide-and-conquer style we know that the problem can be divided into smaller subproblems of the same form—such as a smaller collection.

All algorithms assume that we have many more items of data than processors—that is, $N \gg P$. This allows the amortization of the high fixed overhead of message-passing operations across large messages. I am not investigating algorithms where parallel processors are used to process small amounts of data very quickly, as for instance in real-time computer vision [Web94].

The design decisions also assume that it is many times faster for a processor to perform a computation using local registers than it is to read or write a word to or from local memory, which in turn is many times faster than initiating communication to a different processor. This is true for most current distributed-memory computers, and seems unlikely to change in the near future, given the general physical principle that it's faster to access nearby things than far away ones. Note these times are the latencies to initiate an operation; the bandwidths achievable to local memory and across a network are comparable for some recent architectures [SG97].

11

1.6 Contributions of this Work

The requirements for any new general-purpose parallel model or language include (at least) portability, expressiveness, and performance. It remains unclear how to achieve all three of these goals for the entire field of parallel algorithms, but this dissertation shows that it is feasible if we restrict our scope to divide-and-conquer algorithms (similarly, it can be claimed that HPF achieves these goals for purely data-parallel algorithms).

The other contributions of this dissertation can be summarized as follows:

Models I have defined team parallelism, a new model that allows the efficient parallel implementation of divide-and-conquer algorithms. I have shown the expressive limits of this model in terms of the kinds of divide-and-conquer algorithms that it can efficiently parallelize.

Artifacts I have written a preprocessor for a particular implementation of the team parallel model, which compiles a parallel extension of C into portable C and MPI. In addition, I have written an active load-balancing system based on function shipping, which is built on top of MPI's derived types system, and a library of general-purpose data-parallel routines, incorporating both vector communication and team splitting and combining operations. This library can be used either in conjunction with the preprocessor, or as an independent library called directly from an application program.

Algorithm implementations Using the team parallel model and the Machiavelli components described above, I have implemented the world's fastest two-dimensional Delaunay triangulation algorithm, which is up to three times more efficient than previous implementations. It can also handle realistic, irregular datasets with much less impact on performance than previous implementations. I have also implemented a variety of smaller (although still computationally significant) algorithms, including convex hull, sorting, and matrix multiplication.

1.7 Organization of the Dissertation

The rest of this dissertation is organized as follows.

Chapter 2 describes in more detail the background to the thesis, and related work in the area. I consider the different parallel models that provide nested parallelism, and existing implementations of nested parallelism.

Chapter 3 formally defines my team parallel model, and describes the characteristics of the model and the primitives required for its implementation. I also define axes along which to

define divide-and-conquer algorithms, and classify a range of algorithms according to these axes.

Chapter 4 describes Machiavelli, a particular implementation of the team parallel model. I concentrate in particular on describing design decisions taken during the implementation, and the reasoning behind them.

Chapter 5 evaluates the performance of the Machiavelli system in terms of its primitive operations on three different parallel architectures.

Chapter 6 presents four basic divide-and-conquer algorithms, shows how they can be implemented in Machiavelli, and gives their performance on three parallel architectures.

Chapter 7 describes a major application written using the system, namely a parallel two-dimensional Delaunay triangulation algorithm.

Chapter 8 summarizes the work in the previous chapters, and concludes the dissertation with a look at future directions for research in this area.

14

Chapter 2

Background and Related Work

> *Parallelism is at best an annoying implementation detail.*
> —Ian Angus, WOPA'93

This dissertation builds on previous work in several fields of parallel computing. In Section 2.1 I discuss the concept of nested parallelism, and four general language models that it covers. In Section 2.2 I then describe five different techniques that have been used to implement nested parallelism, in both the language models and for specific algorithms.

2.1 Models of Nested Parallelism

Informally, *nested parallelism* is a general term for the situation where two or more parallel constructs are active at a time, resulting in multiple levels of parallelism being available to the system. More formally, if computation is modelled as directed acyclic graphs, composed of nodes (representing tasks) and edges (representing ordering dependencies between tasks), a nested parallel program is one that results in a series-parallel DAG.

Apart from the higher performance that is typically possible by exposing more parallelism, nested parallelism can also simplify the expression of parallel algorithms, since it allows the direct expression of such concepts as "in parallel, for each vertex in a graph, find its minimum neighbor", or "in parallel, for each row in a matrix, sum the row" [Ble96]. In both of these, the inner actions (finding the minimum neighbor and summing the row) can themselves be parallel. The importance of language support for nested parallelism has been noted elsewhere [TPH92].

Nested parallelism has appeared in parallel computing in at least four general forms: mixed data and control parallelism, nested data parallelism, group-based divide-and-conquer parallelism, and nested loop parallelism. These are presented in rough order of their generality:

15

mixed data and control parallelism being the most general, and nested loop parallelism the most restrictive. I am using "generality" to describe the range of algorithms that can be efficiently expressed by a model; while all of the models are computationally equivalent and could be used to simulate each other, there would be a loss of efficiency when implementing a more general model in terms of a more restrictive one. Also, note that the nesting can be specified either statically (as in the case of nested loop parallelism) or dynamically (as in the case of divide-and-conquer parallelism).

2.1.1 Mixed data and control parallelism

At the most general level, *mixed parallelism* combines data (or loop) parallelism with control (or task) parallelism in a single program. Typically this mixed parallelism is expressed as data parallelism within independent subgroups of processors. This approach has been widely explored in the Fortran field, and was pioneered by the Fx compiler [GOS94]. For applications that contain a mix of data-parallel and control-parallel phases, a static assignment of processors to data-parallel groups, with pipelining of intermediate results between these processor groups, can give better results than either purely data-parallel or purely control-parallel approaches [SSOG93]. More recently, dynamic assignment of processors to groups has been proposed specifically for the case of divide-and-conquer parallelism [SY97] (see Section 2.2.4). Fx also extends nested parallelism by allowing communication between sibling function calls on the same recursive level (that is, allowing the computational DAG to have cross-edges representing communication or synchronization between tasks). This enables the "pipelined task groups" approach pioneered by Fx, and also allows the effective implementation of tree-structured algorithms where leaf nodes must communicate, such as Barnes-Hut [BH86].

2.1.2 Nested data parallelism

Nested data parallelism is an extension of standard (or flat) data parallelism that adds the ability to nest data structures and to apply arbitrary function calls in parallel across such structures [Ble90]. In flat data parallelism a function can be applied in parallel over a collection of elements (for example, performing a sum of a vector or counting the members of a set), but the function itself must be sequential. In nested data parallelism the function can be a parallel function. This allows a natural expression of algorithms with either a fixed or a dynamic level of nesting. As an example, the multiplication of matrices has a fixed level of nesting, with parallelism in the dot-product of rows by columns, and in the application of these independent dot-products in parallel. By contrast, a recursive divide-and-conquer algorithm such as quicksort has a dynamic level of nesting, since each function invocation can potentially result in another two function invocations to be run in parallel, but the number of levels is not known

16

in advance. An important aspect of the nested data-parallel model is that nested sub-structures can have either fixed or variable size. This allows for the efficient execution of algorithms such as quicksort that operate on highly irregular data structures, which would be very difficult to implement efficiently in a flat data-parallel model. As an example, Figure 2.1 shows quicksort expressed in NESL. Here there is parallelism both in the expressions to create `les`, `eql` and `grt`, and in the parallel function calls to create the nested vector `sorted`. NESL will be described further in Section 2.2.1.

```
function quicksort(s) =
if (#s < 2) then s
else
  let pivot = s[#s / 2];
      les = {e in s | e < pivot};
      eql = {e in s | e = pivot};
      grt = {e in s | e > pivot};
      sorted = {quicksort(v) : v in [les, grt]}
  in sorted[0] ++ eql ++ sorted[1];
```

Figure 2.1: Quicksort expressed in NESL, taken from [Ble95]. Compare to Figure 1.1

2.1.3 Divide-and-conquer parallelism

Divide-and-conquer algorithms have long been recognized as possessing a potential source of control parallelism, since the subproblems that are generated can typically be solved independently. Many parallel architectures and parallel programming languages have been designed specifically for the implementation of divide-and-conquer algorithms, as reviewed by Axford [Axf92], with the earliest dating back to 1981 [Pet81, PV81]. Indeed, Axford [Axf90] has proposed to replace the iterative construct in imperative languages and the recursive construct in functional languages with the divide-and-conquer paradigm, precisely because it simplifies the parallelizability of languages.

A complete classification of divide-and-conquer parallelism will be given in Chapter 3. However, to compare previous models it is generally sufficient to consider whether or not they can handle *unbalanced* divide-and-conquer algorithms, in which the divide step does not result in problems of equal sizes. Balanced algorithms execute a fixed number of recursive levels for a given problem size, and hence are easier both to implement and to reason about. They correspond to algorithms that have a fixed computational structure, such as fast Fourier transform. Unbalanced algorithms will tend to go through more recursive levels for some subproblems than for others, and will require some form of load balancing. Unbalanced behavior is typically found in data-dependent algorithms, such as those that use a partitioning operation to divide the data.

The first well-defined model of parallel divide-and-conquer algorithms with an associated implementation was Divacon, presented by Mou [MH88, Mou90]. Divacon is a functional

programming language that uses dynamic arrays and array operations to express divide-and-conquer algorithms. Mou provides a formal definition of the phases of a divide-and-conquer algorithm, and three templates for divide-and-conquer functions: a naive one with a simple test-solve-recurse-combine structure; a more sophisticated variant that adds operations before and after the recursive step, to allow for the pre-adjustment and post-adjustment of data; and a serialized variant that imposes a linear order on the recursive operations. Divacon is limited to balanced algorithms, and was implemented in *Lisp on the SIMD Connection Machine CM-2. Arrays are spread across the machine, and primitive operations such as divide, combine, and indexing are performed in parallel, taking constant time provided the size of the array does not exceed the number of processors.

More recently, Misra [Mis94] defined a balanced binary divide-and-conquer model based on *powerlists*, which are lists whose length is a power of 2. The model extends that of Mou by allowing for very concise recursive definitions of simple divide-and-conquer algorithms, and can be used to derive algebraic proofs of algorithms. Similarly, Achatz and Schulte [AS95] defined a balanced model on sequences, using semantic-preserving transformation rules to map a balanced divide-and-conquer algorithm from an abstract notation into a program specialized for a particular network architecture (for example, an array or a mesh). Again, a functional approach is used, with data-parallel operations being defined as higher-order functions mapped across a data structure.

Cole [Col89] proposed the idea of encapsulating commonly-used parallel algorithmic models in *skeletons*, which can then be filled in with the details of a particular algorithm that matches the model, and used a divide-and-conquer skeleton as one of his main examples. An analysis of its costs on a square mesh of processors is also presented. However, the skeleton is for a purely control-parallel implementation of balanced divide-and-conquer algorithms.

As well as these language-based models of divide-and-conquer parallelism, there have also been machine-based models. Heywood and Ranka [HR92a] proposed the Hierarchical PRAM model as a practical form of PRAM that allows a general model of divide-and-conquer algorithms. The H-PRAM extends the PRAM model to cover group parallelism, defining a collection of individual synchronous PRAMs that operate asynchronously from each other. A partition command is used to divide the processors that compose a PRAM into sub-PRAMs, and to allow the execution of new algorithms on those sub-PRAMs. Two variants of the H-PRAM were proposed. The *private* H-PRAM only allows processors to access memory within their current sub-PRAM—that is, memory is partitioned as well as processors. In the *shared* H-PRAM memory is shared by all the processors, and hence this model is more powerful than the private H-PRAM. These two models can be seen as allowing implementations specialized to distributed-memory or shared-memory architectures. Heywood and Ranka express a preference for the private H-PRAM, since it restricts possible indeterminacy in an algorithm and provides a cleaner programming model. Although the H-PRAM model does not specifically restrict itself to balanced divide-and-conquer algorithms, the only algorithms designed and

18

analysed using it (binary tree and FFT) are balanced [HR92b].

Axford [Axf92] summarized the field of parallel divide-and-conquer models, and considered three choices for the basic parallel data structure: arrays, lists, and sets. Although both lists and sets have advantages for certain application areas (as evidenced by Lisp [SFG$^+$84] and SETL [SDDS86]), Axford concludes that arrays are clearly the easiest to implement, with the efficient implementation of suitable list and set structures being an open problem for distributed-memory architectures. He also notes that although the basic divide-and-conquer construct must itself be side-effect free, there is nothing to prevent the addition of such a construct to a procedural language.

2.1.4 Loop-level parallelism

Although it is the most restricted form of nested parallelism, loop-level parallelism is also the only one that can easily be exploited in existing sequential code. Specifically, loop-level parallelism is potentially available whenever there are independent nested loops. Figure 2.2 shows an example of Fortran extended with an explicit PARALLEL DO loop [TPS$^+$88]. Here the basic unit of independent parallel work is an execution of the inner loop, and a simple parallelization would assign outer loop iterations to individual processors. Nested parallelism becomes useful when the number of processors P is greater than the number of outer iterations N, which would leave the excess processors idle on a non-nested system. By exploiting both levels of parallelism at once, a nested loop-parallel system can assign $M \times N/P$ inner loop executions to each processor [HS91], breaking the "one processor, one outer loop iteration" restriction and balancing the load across the entire machine.

```
PARALLEL DO I = 1,N
  ...
  PARALLEL DO J = 1, M
    ...
  END DO
  ...
END DO
```

Figure 2.2: Example of nested loop parallelism using explicit parallel loop constructs in Fortran. Taken from [HS91].

When using a parallel construct of this nature, the independence of loop iterations can be guaranteed by semantic restrictions on what can be placed inside a loop construct. If we are trying to parallelize a "dusty-deck" program written in a serial language, the compiler must analyze serial nested loop structures to determine whether the iterations are independent, although imperfect nesting can sometimes be restructured into independent loop nests [Xue97].

19

2.2 Implementation Techniques for Nested Parallelism

Historically, five different techniques have been used to implement the models of nested parallelism described in the previous section: flattening nested data parallelism, threading, fork-join parallelism, processor groups, and switched parallelism. No particular order is implied here, although a given language model from the previous section has generally only been implemented using one or two of the techniques. In addition to languages, I will cite implementations of specific algorithms that have used these techniques in a novel way.

2.2.1 Flattening nested data parallelism

Flattening nested data parallelism [BS90, Ble90] has been used exclusively to implement nested data parallelism, as its name implies. The flattening technique was first developed for a partial implementation of the language Paralation Lisp [Sab88], and was then used to fully implement the nested data-parallel language NESL [BHS+94].

NESL is a high-level strongly-typed applicative language, with sequences as the primary data structure. Each element of a sequence can itself be a sequence, giving the required nesting property. Parallelism is expressed through an apply-to-each form to apply a given function to elements of a sequence, and through parallel operations that operate on sequences as a whole. The NESL environment has an interactive compiler similar to the "read-eval-print" loop of a Lisp system, and allows the user to switch between different back-end machines. Combined with the concise high-level nature of the language, this flexibility has resulted in NESL being used extensively for teaching parallel algorithms [BH93], and for prototyping new parallel algorithms [Gre94, BN94, BMT96].

Internally, the representation of a nested sequence in NESL is "flattened" into a single vector holding all of the data, plus additional control vectors (*segment descriptors*) that hold a description of the data's segmentation into nested subsequences. This allows independent functions on nested sequences to be implemented using single data-parallel operations that have been extended to operate independently within the segments of a segmented vector. An important consequence of flattening is the implicit load-balancing that takes place when a possibly irregular segmented structure is reduced to a single vector that can be spread uniformly across the processors of a parallel machine.

The current NESL system compiles and flattens a NESL program into a stack-based vector intermediate language called VCODE [BC90]. The segmented data-parallel VCODE operations are then interpreted at runtime. The performance impact of interpretation is small, because the interpretive overhead of each instruction is amortized over the length of the vectors on which it operates [BHS+94]. The VCODE interpreter itself is written in portable C, and is linked against a machine-specific version of CVL [BCH+93], a segmented data-parallel library that

20

performs the actual computation. CVL has been implemented on Unix workstations and a range of supercomputer architectures [FHS93, Har94, CCC+95]. The design decisions behind CVL assume that the characteristics of the underlying machine are close to those of a PRAM. This is a good match to the SIMD and vector machines that were dominant when CVL was designed, but is no longer true for distributed-memory MIMD machines [Har94]. On these machines computation is virtually free, memory locality is very important, and the startup cost of sending a message is typically high, although inter-processor bandwidth may be quite good. This results in adverse effects from at least three of CVL's design decisions:

1. To enforce the guarantees of work and depth complexity that apply to each NESL primitive, CVL must store vectors in a load-balanced fashion. Since each load-balancing step results in interprocessor communication, this can be a significant cost for algorithms with data structures that change frequently.

2. CVL is a purely data-parallel library, with an implicit synchronization between the processors on every operation; this is not required by NESL, but was chosen to ease the task of implementing CVL. The result is unnecessary synchronization between processors— lock-step execution is typical in a SIMD machine, but unnatural in a MIMD machine.

3. The structure of CVL as a library of vector functions results in poor data locality, because each CVL function is normally implemented as a loop over one or more vectors, which are typically too large to fit in the cache.

Based on the semantics of NESL, Blelloch and Greiner [BG96] have proven optimal time and space bounds for a theoretical implementation of NESL on a butterfly network, hypercube, and CRCW PRAM. These bounds assumes the existence of a fetch-and-add primitive, and are optimal in the sense that for programs with sufficient parallelism the implementation has linear speedup and uses within a constant factor of the sequential space.

After NESL introduced the model of nested data parallelism and the technique of flattening, several other languages adopted both ideas. Proteus [MNP+91] was designed specifically for prototyping parallel programs, and was subsequently extended to handle nested data parallel operations [PP93]. It is a high-level imperative architecture-independent language based on sets and sequences. Being built on top of CVL, it suffers from the same performance problems on distributed-memory machines. Proteus has recently been used to explore two extensions to the basic flattening algorithm. The first technique, work-efficient flattening [PPW95], is used to reduce unnecessary vector replication (and associated loss of work efficiency) that can occur in index operations. The replication is replaced by concurrent read operations, which are randomized to reduce contention. The second technique, piecewise execution [PPCF96], places a bound on the amount of memory that can be used by intermediate vectors. This is important when the nested data-parallel approach exposes "too much" parallelism. The

21

execution layer is extended to enable the execution of data-parallel operations on independent pieces of argument vectors, producing a piece of the result of the operation. Basic blocks of data-parallel operations are then executed inside a loop, with each iteration producting a piece of the result of the basic block. However, the resulting system executes at less than half the speed of the initial data-parallel code.

Chatterjee [Cha91, Cha93] compiled VCODE down to thread-based C code for the Sequent Symmetry, an early shared-memory machine, and developed several compiler optimizations to reconstruct information that had been lost by the NESL compilation phase. Specifically, size inference was used to determine the sizes of vectors, allowing clustering of operations on vectors of the same size. When combined with access inference to detect conflicts between definition and usage patterns, this allows loops representing independent operations to be fused. Storage optimizations were also added to eliminate or reduce storage requirements for some temporary vectors. Note that the Sequent Symmetry came quite close to the ideal of a PRAM, in that the time to access memory was roughly equal to the time taken by an arithmetic operation, and good scalability for several applications was demonstrated to twelve processors.

Sheffler and Chatterjee [SC95] extend nested data parallelism to C++, adding the vector as a parallel collection type, and a `foreach` form to express elementwise parallelism. This allows the direct linkage of other code to the nested data-parallel code. The same basic flattening approach is used to compile the code, and CVL is used as the implementation layer on parallel computers. The dependence on CVL was subsequently removed by Sheffler's AVTL [She96a], a template-based C++ library which defines a vector collection type, some basic operations on vectors, and the ability to parameterize these by operator—for example, calling a scan algorithm on a vector of doubles with an addition operator. This allows the compiler to specialize algorithms at compile time, rather than relying on linking to a library such as CVL that predefines all possible combinations of algorithm, type, and operation. AVTL uses MPI as a communication layer on parallel machines, and can be considered as a more modern and more portable CVL. However, since the code executed at runtime remains essentially the same as that in CVL, the system suffers from the same basic performance problems. In addition, Sheffler found that C++ compilers available on supercomputers were not yet sufficiently mature to handle templates, causing problems on the Connection Machine CM-2, IBM SP-1, and Intel Paragon [SC95, She96a].

The programming language V [CSS95] adds nested data parallelism to a variant of C extended with data-parallel vectors, using elementwise constructs similar to those in NESL and Proteus. Again, flattening is used to compile V into CVL. However, significant restrictions have to be placed on the use of C pointers and side-effecting operations in apply-to-each constructs, resulting in *pure* functions that are similar to those in HPF [HPF93]. V was never fully implemented, being replaced by Fortran90V [ACD$^+$97], which takes the same concepts from V and NESL and applies them to Fortran90. The adoption of Fortran removes the necessity to work around problems with C's pointers, and provides a natural array model to replace

22

the rather artificial sequences. Additionally, it is being implemented on top of a vectorizable library of native Fortran90 functions, in place of CVL. Since it is precompiled, this library is likely to suffer from the same performance limitations as CVL on distributed-memory machines. Fortran90V's design has been finished, but it is not yet implemented. Figure 2.3 shows the basic divide-and-conquer quicksort algorithm from Chapter 1 expressed in Proteus, V, and Fortran90V.

Sipelstein [Sip] has proposed to improve the performance of CVL through the use of *lazy vectors*. This involves the use of run-time and compile-time knowledge of computation vs communication costs to decide the right time to perform a communication step. For example, the CVL `pack` operation extracts selected elements from a source vector to create a destination vector. As explained above, the implicit load balancing of the destination vector will cause communication as the elements are moved. In a function performing many packs, such as quicksort, it is worth postponing the packs until several can be performed at once as a single operation. For intermediate computation steps, a flag is added to every element in the resulting lazy vector to indicate whether or not it has been conceptually packed out. Packed-out elements take no further part in the computation, but must still be checked on every step, thereby introducing extra computational cost. A run-time system is used to decide when to perform a communication step, based on compile-time knowledge of the machine's parameters. The assumption is that communication is much more expensive than computation, which is true for current distributed-memory computers.

2.2.2 Threads

A second way to implement nested data parallelism is to postpone the interpretation of the nesting until run time by using threading. For example, Narlikar and Blelloch [NB97] describe a scheduler that can perform this task for a NESL-like language running on an SMP. Iterations of inner loops are grouped into chunks and executed as separate tasks, each within a thread. The execution order of the tasks is matched to the execution order that a serial implementation would use—that is, a depth-first traversal of the computation DAG. The scheduler is provably time and space efficient, and in practice uses less memory than previous systems whilst achieving equivalent speedups. Applications using this scheduler must currently be hand-coded in a continuation-passing style.

Similarly, Engelhardt and Wengelborn [EW95] have defined a nested data parallel extension to the Adl language, with a prototype implementation based on nodal multithreading. The target platform is the CM-5, a distributed-memory multicomputer. Adl allows nested data structures to be partitioned using a user-defined function across the processors, and there is therefore the potential for contention when a processor must cooperate with other processors to update segments of two or more data structures at once. A threaded abstract machine model,

23

```
function qsort (s)
  return
    if #s < 2 then s;
    else let pivot = arb (s);
             les = [e in s | e <  pivot : e];
             eql = [e in s | e == pivot : e];
             grt = [e in s | e >  pivot : e];
             sorted = [r in [lesser, greater] : qsort(s)];
         in sorted[1] ++ eql ++ sorted[2];


pure double qsort (double s[*])[*]
{
  double pivot;
  double les[*], eql[*], grt[*];
  double result[*][*];

  if ($s < 2) return s;

  pivot = s[rand($s)];
  les = [x : x in s : x < m];
  eql = [x : x in s : x == m];
  grt = [x : x in s : x > m];
  sorted = [qsort (subseq) : subseq in [les, grt]];
  return sorted[0] >< eql >< sorted[1];
}


PURE RECURSIVE FUNCTION qsort (s) RESULT (result)
  REAL, DIMENSION (:), IRREGULAR, INTENT (IN) :: s
  REAL, DIMENSION (:), IRREGULAR            :: result, les, eql, grt
  REAL                                      :: pivot

  IF (SIZE(s) < 2) THEN
    result = s
  ELSE
    pivot = s(rand(SIZE(s)))
    les = (< e : e = s : e < pivot >)
    eql = (< e : e = s : e = pivot >)
    grt = (< e : e = s : e > pivot >)
    sorted = flatten ((< qsort (subseq) : subseq = (< les, grt >) >))
  ENDIF
END FUNCTION qsort
```

Figure 2.3: Quicksort expressed in three nested data-parallel languages: Proteus (taken from [GPR+96]), V (taken from [CSS95]), and F90V (derived from [ACD+97]). Note that the Proteus version is polymorphic, while the V and F90V versions are specialized for doubles.

combined with the CM-5's active messages layer, is used to avoid deadlock by assigning a thread to execute each function invocation on an inner nested data structure.

Merrall [Mer96] uses a combination of run-time interpretation and processor groups (see Section 2.2.4) to handle nested data-parallelism. As explained in Section 2.2.1, flattening was originally developed to implement a subset of Paralation Lisp. The paralation model extends Lisp with data-parallel *fields* (objects) and elementwise parallel operators. Nested parallelism comes from the ability to create fields of fields, and to nest elementwise operations to manipulate them. However, flattening also imposes strong type constraints on the nested expressions, so that they can be executed by a single data-parallel operation. Merrall's Paralation EuLisp extends the Lisp variant EuLisp with nested parallelism, and removes these typing restrictions by dealing with the nested parallelism at run time rather than at compile time. The implementation on an SIMD MasPar machine uses an independent bytecode interpter on every processor, controlled by a Lisp process running on the host. When a nested operation is encountered, the SIMD processors are divided into *sets*, each of which is responsible for a nested sub-structure. By downloading bytecode for different functions to different sets, the type restrictions on a nested operation can be relaxed. However, no examples are given of applications which would benefit from this extension to the nested data-parallel model. The performance impact is also severe, for two main reasons. First, the process of downloading code to different sets must be sequentialized, resulting in an overhead proportional to the number of substructures in a nested data structure. Second, to achieve MIMD parallelism the processors must run a bytecode interpreter. This executes a loop of about 50 SIMD instructions broadcast from the host for every simulated MIMD instruction.

2.2.3 Fork-join parallelim

Most implementations of parallel divide-and-conquer languages have used the *fork-join* model, relying on multiple independent processors running sequential code to achieve parallelism. In this approach, a single processor is active at the beginning of execution. The code on an active processor forks off function calls onto other, idle processors at each recursive level. Once every processor is active, the inter-processor recursion is replaced by a standard sequential function call. After the recursion has completed, processors return their results to the processor that forked them, so that the original processor ends up with the final result

Fork-join parallelism is purely control parallel, and as we saw in Chapter 1 this limits the efficiency of the resulting program, since at both ends of the recursion tree all but one of the processors will be inactive. On distributed-memory machines it also restricts the maximum problem size to that which can be held on one processor, which eliminates another reason for using a parallel computer. Nevertheless, its simplicity has made it a popular choice in implementing divide-and-conquer languages [MS87, CHM+90, PP92, DGT93, Erl95].

25

2.2.4 Processor groups

The use of parallelism both within and between groups of processors to implement nested parallelism has been widely explored. As noted in Section 2.1.1, the Fx compiler [GOS94] supports full mixed parallelism in an HPF variant, albeit with statically-specified processor groups. Subhlok and Yang [SY97] have recently extended this with dynamic processor groups, enabling the direct implementation of divide-and-conquer algorithms, and a similar extension has been approved for the proposed HPF2 standard. To interface with the existing Fx compiler, the implementation adds processor mappings to translate from virtual processor numbers to physical processors, and a stack of *task regions*, which are the syntactic equivalent of processor groups. As an example, Figure 2.4 shows quicksort expressed in Fx. However, performance improvements over a basic data-parallel approach are relatively modest.

```
SUBROUTINE qsort(a, n)
  INTEGER n, a(n)
  INTEGER pivot, nLess, p1, p2

  if (n .eq. 1) return
  if (NUMBER_OF_PROCESSORS() .eq. 1) then
    call qsort_sequential(a, n)
    return
  endif
  pivot = pick_pivot(a, n)
  nLess = count_less_than_pivot(a, n)
  call compute_subgroup_sizes(n, nLess, p1, p2)
  call qsort_helper(a, n, nLess, n-nLess, p1, p2, pivot)
END

SUBROUTINE qsort_helper(a, n, nLess, nGreaterEq, p1, p2, pivot)
  INTEGER n, a(n), nLess, nGreaterEq, p1, p2, pivot
  TASK_PARTITION qsortPart:: lessG(p1), greaterEqG(p2)
  INTEGER aLess(nLess), aGreaterEq(nGreaterEq)
  SUBGROUP(lessG):: aLess
  SUBGROUP(greaterEqG):: aGreaterEq

  BEGIN TASK_REGION qsortPart
    CALL pick_less_than_pivot(aLess, nLess, a, n, pivot)
    CALL pick_greater_equal_to_pivot(aGreaterEq, nGreaterEq, a, n, pivot)
    ON SUBGROUP lessG
      call qsort(aLess, nLess)
    END ON SUBGROUP
    ON SUBGROUP greaterEqG
      call qsort(aGreaterEq, nGreaterEq)
    END ON SUBGROUP
    call merge_result(a, aLess, aGreaterEq)
  END TASK_REGION
END
```

Figure 2.4: Quicksort expressed in Fx extended with dynamically nested paralellism (taken from [SY97]). The auxiliary functions qsort_sequential, pick_pivot, count_less_than_pivot, compute_subgroup_sizes, pick_greater_equal_to, and merge_result are not shown.

26

PCP [GWI91, Bro95] was the first language to explicitly differentiate between the *fork-join* model, in which an initial processor spawns code on other processsors as necessary (that is, purely control-parallel), and the *split-join* model, in which all the processors begin in one team, which is subdivided as necessary (that is, mixed control and data parallelism). PCP is implemented as a collection of simple parallel extensions to C, and a conscious effort has been made to emphasize speed. In particular, the language is designed to be *reducible*, in that if certain operations necessary for a particular architecture reduce to null operations in more sophisticated hardware, the overhead of the operations should also be eliminated. Thus, PCP compiles to serial code on workstations, to shared-memory code on SMPs, and to code that uses active messages for remote fetch on MPPs.

Within PCP, data-parallel behavior is provided by the `forall` construct, while control-parallel behavior is provided by `split` (to create different tasks), and `splitall` (to create multiple instances of the same task). C's storage classes are extended with `shared` memory, which is accessible by all processors, and `teamprivate` memory, which is accessible only by the members of a particular team (note the similarity to the H-PRAM variants discussed in Section 2.1.3). PCP's design was influenced by the architecture of early SMP's, to which it was widely ported—it depends on hardware support for finely-interleaved shared memory, or remote fetch via active messages on MPPs. In particular, globally addressable shared memory is required because PCP allows arbitrary code (including code that accesses shared memory) inside `split` blocks.

Recently, Heywood et al[CCH95] have begun investigating implementation techniques for parts of the H-PRAM model, analyzing the Hilbert indexing scheme (sometimes referred to as Peano indexing) in terms of routing requirements and the longest path between two nodes of a sub-group. [CCH96] describes an algorithm that subdivides a group of processors in a 2D mesh, constructing a binary synchronization tree for each group. This is similar to the algorithm that must be used to subdivide processor groups in e.g. MPI [For94], but is more advanced in that it accounts for processor locality. Simulation results are used to verify the performance analysis on a mesh, using both row/column and Hilbert indexing schemes.

Kumaran and Quinn [KQ95] implement a version of the parallel divide-and-conquer template defined by Mou and Hudak. In addition to the Divacon restriction to balanced divide-and-conquer algorithms, their model is further restricted to only support those problems that use a left-right division (that is, there is no data movement involved in the split of a problem). This considerably simplifies the implementation—it is an "embarassingly divisible" divide-and-conquer equivalent of "embarassingly parallel" algorithms. Basic communication functions are provided for data-parallel operations within processor groups. When the code has recursed down to a single processor, a serial function is called, but it is unrolled into an iterative loop rather than being run recursively. The system is hand-coded, using Dataparallel C [HQ91] on a Sequent Symmetry, and C plus CMMD on a Thinking Machines CM-5. For the restricted range of algorithms possible, good speedups are achieved on both machines.

27

Algorithm implementations

In addition to programming languages, processor groups have been used for one-off implementations of several important parallel algorithms. Gates and Arbenz [GA94] used group parallelism to implement a balanced divide-and-conquer algorithm for the symmetric tridiagonal eigenvalue problem. This algorithm is interesting in that more work is done at the root of the tree than at the leaves. One result is that, for problem and machine sizes of interest, any final load imbalance on nodes is relatively small, and hence no active load-balancing system is necessary. An implementation on the Intel Paragon showed good scalability to 64 processors, and achieved one-third parallel efficiency (that is, efficiency compared to the best sequential algorithm). The authors conclude that the divide-and-conquer approach allows them to contradict an earlier paper [IJ90] which stated that the potential for parallelizing this class of algorithms was poor.

Ma et al [MPHK94] describe the use of a divide-and-conquer algorithm for ray-traced volume rendering and image composition on MIMD computers. As with other divide-and-conquer rendering algorithms, independent ray-tracing is done on processors at the leaves of the recursion. However, nested parallelism is exploited by using all of the processors to perform the image composition phase, rather than halving the number of processors involved at each level on the way back up the recursion tree. The algorithm was implemented using two different message-passing systems, CMMD on the CM-5 and PVM on a network of workstations. It showed good scalability on regular datasets, but fared less well on real-world data sets where many voxels are empty.

Barnard [BPS93] implemented a parallel multilevel recursive spectral bisection algorithm using processor groups, which he termed *recursive asynchronous task teams*. Team control was abstracted into `split` and `join` routines. Asynchronous behavior was emphasized, and several practical implementations of theoretical CRCW PRAM algorithms were developed as substeps (for example, finding a maximal independent set, and identifying connected components). However, the algorithm was specialized for the Cray T3D, using its cache-coherent communication primitives `shmem_get` and `shmem_put` to emulate the behavior of an SMP.

2.2.5 Switched parallelism

Switched parallelism is a simplified variant of full group parallelism. So far it has been used only for particular algorithm implementations, rather than as part of a full language. The algorithms implemented fit the divide-and-conquer model of nested parallelism.

Full group parallelism subdivides the initial group into smaller and smaller groups as the algorithm recurses, and uses flat data-parallel operations within each control-parallel group, until each group contains a single processor. In switched parallelism, the processors stay in

the initial data-parallel group for several levels of recursion, and then switch to control-parallel serial code once subproblems are small enough to fit on a single processor. Group operations must now cope with the multiple nested calls present at a given level of recursion, either by using nested data-parallel operations or by serializing over the calls.

An ad-hoc variant of switched parallelism was used by Bischof et al [BHLS+94] for a divide-and-conquer symmetric invariant subspace decomposition eigensolver. This algorithm uses purely data-parallel behavior at the upper levels of recursion, solving subproblems using the entire processor mesh. Subproblems that are too small for this to be profitable are handled in a control-parallel end-game stage, which solves subproblems on individual nodes. The implementation showed good scalability to 256 processors on the Intel Delta and Paragon, although it was highly tuned for the particular algorithm and architecture.

Chakrabarti et al [CDY95] modeled the performance of data parallel, mixed parallel, and switched parallel divide-and-conquer algorithms, and validated these figures by carrying out experiments on a particular architecture. In particular, they provide formulae and upper bounds for the benefits of mixed parallelism over data parallelism. They conclude that switched parallelism provides most of the benefits of full mixed parallelism for many problems. However, they concentrate on balanced algorithms, and ignore the cost of communication within a task, thereby favoring algorithms with high computational costs.

Goil et al [GAR96] propose *concatenated parallelism*, which can be viewed as a particular implementation strategy for switched parallelism. Rather than dividing processors into groups as in the mixed parallel approach (and hence moving data between processors), they subdivide the data within processors, and serialize over the function calls on each recursive level, performing what is effectively a breadth-first traversal of the computational DAG. The switch from executing data-parallel code as part of a single group to the control-parallel execution of serial code takes place when the algorithm has recursed to the point where there are several subtasks for every processor. Concatenated parallelism avoids data movement between processors to redistribute subtasks on every recursive step, and is therefore a useful technique when the communication time to perform this redistribution is significant compared to the computation time. Examples given in [GAR96] include quicksort, quickhull, and quadtree construction.

Concatenated parallelism also avoids active load-balancing, relying instead on redistributing the tasks across processors to get an approximately equal balance just before the switch to control-parallel serial code. A disadvantage of this approach is that in practice tasks of the same size are unlikely to take the same amount of time in an unbalanced and data-dependent divide-and-conquer algorithm. For example, consider two quicksort subtasks of the same size, with one picking a sequence of pathological pivots while the other picks a sequence of perfect pivots. Concatenated parallelism therefore relies on there being "enough" subtasks per processor that any variations in the actual time taken to perform an individual subtask will tend to average out. However, this involves spending more levels of recursion in parallel code,

29

which will be slower than the serial code due to communication overhead. There is therefore a tradeoff between load imbalance and excessive parallel overhead.

Concatenated parallelism has several other disadvantages. First, communication within a phase is more expensive, due both to the lack of locality in the communication network (the entire machine is involved on every communication step, rather than clusters of processors), and to the presence on a single processor of data from every function call at each level. The latter problem can be partially alleviated by combining communication from multiple serialized function calls. Second, there is an implicit assumption that the dividing operation does not change the total amount of data on a node in a data-dependent way. This holds true for simple quicksort, which only partitions the data, but is not the case for more sophisticated algorithms such as Chan's pruning quickhull [CSY95], which discards data elements to reduce the size of subproblems. This would lead to load imbalances between nodes in a concatenated-parallel implementation of the algorithm.

Finally, and most importantly, concatenated parallelism places severe restrictions on the algorithms that can be efficiently expressed in the model. Specifically, its distributed data structures cannot efficiently support the concept of indexing. While simple data-parallel expressions and filtering operations (such as selecting out all elements less than a pivot) are cheap, anything that requires indexing (such as extracting a specific element, permuting via an index vector, or fetching via an index vector) will be expensive to implement, in terms of either time or space. To see why this is so, consider the structure of a vector after several recursive levels of an algorithm: it is not balanced across the processors, but can have an arbitrary number of elements on each processor. A processor cannot therefore use simple math to compute the processor and offset that a particular index location in the vector corresponds to. There are two possible solutions. The first is to create a replicated index vector on each processor, containing for every element the processor and offset that it maps to. This vector takes $O(P)$ time to construct and only $O(1)$ time to use, but is costly in terms of space, since it replicates on each processor a vector of length n that was previously spread across all processors. The second solution is for each processor to store the number of elements held by every other processor. Whenever an indexing operation is required, it then performs a binary search on this array of element counts to find the processor holding the required element. This requires only $O(P)$ space on each processor, but $O(\log P)$ time for each indexing operation.

2.3   Summary

Of the four models that can be used to implement nested parallelism, flattening nested data parallelism works well for machines similar in nature to a PRAM—that is, most SIMD and vector multiprocessor machines. These machines typically have high memory bandwidth and no caches, which allows them to ignore the potential performance bottlenecks of a flattened

approach, namely high bandwidth requirements due to implicit load-balancing, and the lack of locality due to individual data-parallel operations. However, for current distributed-memory machines these problems become much more severe, and a general, portable solution would require a combination of sophisticated compiler analysis and data representation optimizations such as lazy vectors. A significant advantage of flattening is the portability of the resulting implementation layer, which can be expressed (albeit at the cost of a loss of locality) as a library of segmented data-parallel operations.

Threading works well as an implementation approach for nested parallelism on shared-memory multiprocessors, and distributed-memory machines where a globally-addressed memory can be simulated by the use of active messages. However, neither of these approaches are portable to the majority of distributed-memory machines, which lack both a standard threads system and a standard active messages library.

The most promising implementation approach in terms of widespread portability, efficiency, and generality is group-based parallelism. It has been shown to work well for specific algorithms on a range of machines, but previous languages with support for group-based parallelism have lacked portability and the ability to handle unbalanced divide-and-conquer algorithms.

Chapter 3

The Team-Parallel Model

In time, fantastic myths give way to bloodless abstractions.
—Edward Said

This chapter is devoted to models of algorithms, and approaches to programming them in parallel. In Section 3.1 I discuss the divide-and-conquer paradigm, and define the differences between divide-and-conquer algorithms that are important when they are to be implemented on parallel machines. Then in Section 3.2 I define the team parallelism model, and show how it can implement all of the previously discussed algorithms.

3.1 Divide-and-Conquer Algorithms

Informally, the divide-and-conquer strategy can be described as follows [Sto87]:

> To solve a large instance of a problem, break it into smaller instances of the same problem, and use the solutions of these to solve the original problem.

Divide-and-conquer is a variant of the more general *top-down* programming strategy, but is distinguished by the fact that the subproblems are instances of the same problem. Divide-and-conquer algorithms can therefore be expressed recursively, applying the same algorithm to the subproblems as to the original problem. As with any recursive algorithm, we need a *base case* to terminate the recursion. Typically this tests whether the problem is small enough to be solved by a direct method. For example, in quicksort the base case is reached when there are 0 or 1 elements in the list. At this point the list is sorted, so to solve the problem we just return the input list.

33

Apart from the base case, we also need a *divide phase*, to break the problem up into sub-problems, and a *combine phase*, to combine the solutions of the subproblems into a solution to the original problem. As an example, quicksort's divide phase breaks the original list into three lists—containing elements less than, equal to, and greater than the pivot—and its combine phase appends the two sorted subsolutions on either side of the list containing equal elements.

This structure of a base case, direct solver, divide phase, and combine phase can be generalized into a template (or skeleton) for divide-and-conquer algorithms. Pseudocode for such a template is shown in Figure 3.1. The literature contains many variations of this basic template [Mou90, Axf92, KQ95].

```
function d_and_c (p)
   if basecase (p)
   then
      return solve (p)
   else
      (p₁, ..., pₙ) = divide (p)
      return combine (d_and_c (p₁), ..., d_and_c (pₙ))
   endif
```

Figure 3.1: Pseudocode for a generic n-way divide-and-conquer algorithm

Using this basic template as a reference, there are now several axes along which we can differentiate divide-and-conquer algorithms, and in particular their implementation on parallel architectures. In the rest of this section I will describe these axes and list algorithms that differ along them. Four of these axes—branching factor, balance, data-dependence of divide function, and sequentiality—have been previously described in the theoretical literature. However, the remaining three—data parallelism, embarassing divisibility, and data-dependence of size function—have not to my knowledge been defined or discussed, although they are important from an implementation perspective.

3.1.1   Branching Factor

The *branching factor* of a divide-and-conquer algorithm is the number of subproblems into which a problem is divided.

This is the most obvious way of classifying divide-and-conquer algorithms, and has also been referred to as the *degree of recursion* [RM91]. For true divide-and-conquer algorithms the branching factor must be two or more, since otherwise the problem is not being divided.

34

Many common divide-and-conquer algorithms, including most planar computational geometry algorithms, have a branching factor of two. Note that the quicksort seen in Chapter 1 also has a branching factor of two, since even though the initial problem is divided into three pieces, only two of them are recursed on. Some algorithms dealing with a two-dimensional problem space (for example, n-body algorithms on a plane) have a branching factor of four. Strassen's matrix multiplication algorithm [Str69] has a branching factor of seven.

Some early models of parallel divide-and-conquer restricted themselves purely to binary algorithms, since this simplifies mapping the algorithm to network models such as the hypercube (see Section 3.1.2). The case where the branching factor is not naturally mapped onto the target machine—for example, an algorithm with a branching factor of four on a twelve-processor SGI Power Challenge, or Strassen's seven-way matrix multiplication algorithm on almost any machine—is similar to that of unbalanced or near-balanced algorithms (see below) in that some form of load-balancing is required.

Given this support, the branching factor has little effect on the parallel implementation of a divide-and-conquer algorithm, beyond the obvious added complexity of having more things to keep track of for higher branching factors. There can be performance advantages to offset this complexity, in that a higher branching factor could mean that the base case will be reached in fewer recursive steps. If recursive steps are relatively slow compared to the base case, it can be worth the extra algorithmic complexity of doing a multi-way split in order to reduce the number of recursive steps. For example, Sanders and Hansch [SH97] sped up a group-parallel quicksort by a factor of approximately 1.2 by doing a four-way split based on three pivots.

All algorithms described in this thesis have a constant branching factor, and the parallel constructs supplied by the Machiavelli system currently only support algorithms with a constant branching factor. This allows the recursive calls to be compiled into straight-line code. If the number of calls is not constant but is determined by either the value or the size of the data (for example, an algorithm that divides its input of size N into $\sqrt{N}$ sub-problems, each of size $\sqrt{N}$ each, as studied by [ABM94]), the straight-line code would have to be replaced by a looping construct. The team management code would also become more complex, as constants are replaced by variables, but there would be no fundamental problem to supplying this functionality. Perhaps the biggest obstacle would be supplying a general and readable language syntax that could express a variable branching factor.

3.1.2 Balance

A divide-and-conquer algorithm is *balanced* if it divides the initial problem into equally-sized subproblems.

This has typically been defined only for the case where the sizes of the subproblems sum to the size of the initial problem, for example in a binary divide-and-conquer algorithm dividing

35

a problem of size N into two subproblems of size $N/2$ [GAR96]. Mergesort, and balanced binary tree algorithms such as reductions and prefix sums, are examples of balanced divide-and-conquer algorithms.

The advantage of balanced divide-and-conquer algorithms from an implementation standpoint is that they typically require no load balancing. Specifically, for all existing programming models, and assuming that the number of processors is a power of the branching factor of the algorithm and that the amount of work done depends on the amount of data, all of the processors will have equal amounts of work, and hence will need no load balancing at runtime. For example, a binary balanced divide-and-conquer algorithm can be easily mapped onto a number of processors equal to a power of two. Furthermore, on hypercube networks the communication tree of a divide-and-conquer algorithm reduces to communication between processors on a different dimension of the hypercube on every recursive step. This has led to a wide variety of theoretical papers on the mapping of divide-and-conquer algorithms to hypercubes [Cox88, HL91, WM91, Gor96, MW96].

In order to implement an unbalanced divide-and-conquer algorithm, some form of runtime load balancing must be used. This added complexity appears to be one of the main reasons why very few programming models that support unbalanced divide-and-conquer algorithms have been implemented [MS87, GAR96].

Near-balance

> A particular instantiation of a balanced divide-and-conquer algorithm is *near-balanced* if it cannot be mapped in a balanced fashion onto the underlying machine at run time.

Near-balance is another argument for supporting some form of load balancing, as it can occur even in a balanced divide-and-conquer algorithm. This can happen for two reasons. The first is that the problem size is not a multiple of a power of the branching factor of the algorithm. For example, a near-balanced problem of size 17 would be divided into subproblems of sizes 9 and 8 by a binary divide-and-conquer algorithm. At worst, balanced models with no load-balancing must pad their inputs to the next higher power of the branching factor (i.e., to 32 in this case). This can result in a slowdown of at most the branching factor.

The second reason is that, even if the input is padded to the correct length, it may not be possible to evenly map the tree structure of the algorithm onto the underlying processors. For example, in the absence of load-balancing a balanced binary divide-and-conquer problem on a twelve-processor SGI Power Challenge could efficiently use at most eight of the processors. Again, this can result in a slowdown of at most the branching factor.

To my knowledge, near-balance has not previously been studied. However, in terms of implementation a halving (or worse) of efficiency cannot be ignored.

3.1.3 Embarassing divisibility

> A balanced divide-and-conquer algorithm is *embarassingly divisible* if the divide step can be performed in constant time.

This is the most restrictive form of a balanced divide-and-conquer algorithm, and is a divide-and-conquer case of the class of "embarassingly parallel" problems in which no (or very little) inter-processor communication is necessary. In practice, embarassingly divisible algorithms are those in which the problem can be treated immediately as two or more subproblems, and hence no extra data movement is necessary in the divide step. For the particular case of an embarassingly divisible binary divide-and-conquer algorithm, Kumaran and Quinn [KQ95] coined the term *left-right* algorithm (since the initial input data is merely treated as left and right halves) and restricted their model to this class of algorithms. Examples of embarassingly divisible algorithms include dot product and matrix multiplication of balanced matrices.

Embarassingly divisible algorithms are a good match to hypercube networks. As a result Carpentieri and Mou [CM91] proposed a scheme for transforming odd-even divisions—which subdivide a sequence into its odd and even elements—into left-right divisions.

3.1.4 Data dependence of divide function

> A divide-and-conquer algorithm has a *data-dependent divide function* if the relative sizes of subproblems is dependent in some way on the input data.

This subclass accounts for the bulk of unbalanced algorithms. For example, a quicksort algorithm can choose an arbitrary pivot element with which to partition the problem into subproblems, resulting in an unbalanced algorithm with a data-dependent divide function. Alternatively, it can use the median element, resulting in a balanced algorithm with a divide function that is independent of the data. Rabhi and Manson [RM91] use the term "irrregular" to describe algorithms with data-dependent splits, while Mou [Mou90] uses the term "non-polymorphic" (in the sense that the algorithm is not independent of the values of the data), and Goil et al [GAR96] use the term "randomized". Since all of these terms are more commonly used to describe other properties of algorithms and models, I chose a more literal description of the property.

Note that an algorithm may be unbalanced *without* having a data-dependent divide function, if the structure of the algorithm is such that the sizes of the subproblems bear a predictable but unequal relationship to each other. The Towers of Hanoi algorithm and computing the nth Fibonacci number have both been used as examples of such algorithms in the parallel divide-and-conquer literature [RM91, ABM94]. Of course, neither problem is likely to find many practical uses. Algorithms of this sort are typically better programmed by means of dynamic programming [AHU74], rather than divide-and-conquer.

37

3.1.5   Data dependence of size function

An unbalanced divide-and-conquer algorithm has a *data-dependent size function*
if the total size of the subproblems is dependent in some way on the input data.

This definition should be contrasted with that of the data-dependent divide function, in which the total amount of data at a particular level is fixed, but its partitioning is not. Algorithms having a data-dependent size function form a subclass, since they implicitly result in a data-dependent divide. The model is of algorithms that either add or discard elements based on the input data. In practice we shall only see algorithms that discard data, in an effort to further prune the problem size.

For example, a two-way quicksort algorithm—which divides the data into two pieces, containing those elements that are less than, and equal to or greater than, the pivot—does not have a data-dependent size function, because the sum of the sizes of the subtasks is equal to the size of the original task. However, a three-way quicksort algorithm—which divides the data into those elements less than, equal to, and greater than the pivot—does have a data-dependent size function, because it only recurses on the elements less than and greater than the pivot. A more practical example is the convex hull algorithm due to Chan et al [CSY95] (see Chapter 6), which uses pruning to limit imbalance, by ensuring that a subproblem contains no more than three-quarters of the data in the original problem.

Most algorithms that do not have a data-dependent size function fit the simpler definition that the sum of the sizes of the subtasks equals the size of the original task. However, the broader definition allows the inclusion of algorithms that reduce or expand the amount of data by a predetermined factor.

Divide-and-conquer algorithms that have a data-dependent size function can be harder to implement than those that have just a data-dependent divide function, since load imbalances can arise in the total amount of data on a processor rather than simply in its partitioning. This is a particular problem for the concatenated parallel model, as was shown in Chapter 2.

3.1.6   Control parallelism or sequentiality

A divide-and-conquer algorithm is *sequential* if the subproblems must be executed
in a certain order.

The parallelization of sequential divide-and-conquer algorithms was first defined and studied by Mou [Mou90]. Ordering occurs when the result of one subtask is needed for the computation of another subtask. A simple example of an ordered algorithm given by Mou is a naive divide-and-conquer implementation of the scan, or prefix sum, algorithm. In this case

the result from the first subtree is passed to the computation of the second subtree, so that it can be used as an initial value. Note that this is a somewhat unrealistic example, since on a parallel machine scans are generally treated as primitive operations [Ble87] and implemented using a specialized tree algorithm between the processors.

The ordering of subproblems in a sequential divide-and-conquer algorithm eliminates the possibility of achieving control parallelism through executing two or more subtasks at once, and hence any speedup must be achieved through data parallelism. Only Mou's Divacon model [Mou90] has provided explicit support for sequential divide-and-conquer algorithms.

3.1.7 Data parallelism

A divide-and-conquer algorithm is *data parallel* if the test, divide, and combine operations do not contain any serial bottlenecks.

As well as the control parallelism inherent in the recursive step, we would also like to exploit data parallelism in the test, divide and combine phases. Again, if this parallelism is not present, the possible speedup of the algorithm is severely restricted. Data parallelism is almost always present in the divide stage, since division is typically structure-based (for example, the two halves of a matrix in matrix multiplication) or value-based (for example, elements less than and greater than the pivot in quicksort), both of which can be trivially implemented in a data-parallel fashion. The data parallelism present before a divide stage is also called "preallocation" by Acker et al [ABM94].

As an example of an algorithm with a serializing step in its combine phase, consider the standard serial mergesort algorithm. Although the divide phase is data-parallel (indeed, separating the input into two left and right halves is embarassingly divisible), and the two recursive subtasks are not ordered and hence can be run in parallel, the final merge stage must serialize in order to compare sequential elements from the two sorted subsequences. Using a parallel merge phase, as in [Bat68], eliminates this bottleneck.

3.1.8 A classification of algorithms

Having developed a range of characteristics by which to classify divide-and-conquer algorithms, we can now list these characteristics for several useful algorithms, as shown in Table 3.1.

39

	Branching factor	Balanced?	Embarrassingly divisible?	Data-dependent divide function?	Data-dependent size function?	Control parallel?	Data parallel?
Two-way quicksort	2	×	×	√	×	√	√
Three-way quicksort	2	×	×	√	√	√	√
Quickhull	2	×	×	√	√	√	√
2D geometric separator	2	√	×	×	×	√	√
Adaptive quadrature	2	√	√	×	×	√	×
Naive matrix multiplication	8	√	√	×	×	√	√
Strassen's matrix multiplication	7	√	√	×	×	√	√
Naive merge sort	2	√	√	×	×	√	×

Table 3.1: Characteristics of a range of divide-and-conquer algorithms. Note that quicksort and quickhull can be converted into balanced algorithms by finding true medians.

3.2 Team Parallelism

In this section I describe *team parallelism*, my parallel programming model for divide-and-conquer algorithms. I concentrate on describing the characteristics of the model: Machiavelli, a particular implementation of team parallelism, is described in Chapter 4.

Team parallelism is designed to support all of the characteristics of the divide-and-conquer algorithms shown in Table 3.1. There are four main features of the model:

1. Asynchronous subdividable teams of processors.

2. A collection-oriented data type supporting data-parallel operations within teams.

3. Efficient serial code executing on single processors.

4. An active load-balancing system.

All of these concepts have previously been implemented in the context of divide-and-conquer algorithms. However, team parallelism is the first to combine all four. This enables the efficient parallelization of a wide range of divide-and-conquer algorithms, including both balanced and unbalanced examples. I will define each of these features and their relevance to divide-and-conquer algorithms, and discuss their implications, concentrating in particular on message-passing distributed memory machines.

40

3.2.1 Teams of processors

As its name suggests, team parallelism uses *teams* of processors. These are independent and distinct subsets of processors. Teams can divide into two or more subteams, and merge with sibling subteams to reform their original parent team. This matches the behavior of a divide-and-conquer algorithm, with one subproblem being assigned per subteam.

Sibling teams run asynchronously with respect to each other, with no communication or synchronization between teams. This matches the independence of recursive calls in a divide-and-conquer algorithm. Communication between teams happens only when teams are split or merged.

Discussion The use of smaller and smaller teams has performance advantages for implementations of team parallelism. First, assuming that the subdivision of teams is done on a locality-preserving basis, the smaller subteams will have greater network locality than their parent team. For most interconnection network topologies, more bisection bandwidth is available in smaller subsections of the network than is available across the network as a whole, and latency may also be lower due to fewer hops between processors. For example, achievable point-to-point bandwidth on the IBM SP2 falls from 34 MB/s on 8 processors, to 24 MB/s on 32 processors, and to 22 MB/s on 64 processors [Gro96]. Also, collective communication constructs in a message-passing layers typically have a dependency on the number of processors involved. For example, barriers, reductions and scans are typically implemented as a virtual tree of processors, resulting in a latency of $O(\log P)$, while the latency of all-to-all communication constructs has a term proportional to P, corresponding to the point-to-point messages on which the construct is built [HWW97].

In addition, the fact that the teams run asynchronously with respect to one another can reduce peak inter-processor bandwidth requirements. If the teams were to operate synchronously with each other, as well as within themselves, then data-parallel operations would execute in lockstep across all processors. For operations involving communication, this would result in total bandwidth requirements proportional to the total number of processors. However, if the teams run asynchronously with respect to each other, and are operating on an unbalanced algorithm, then it is less likely that all of the teams will be executing a data-parallel operation involving communication at the same instant in time. This is particularly important on machines where the network is a single shared resource, such as a bus on a shared-memory machine.

Since there is no communication or synchronization between teams, all data that is required for a particular function call of a divide-and-conquer algorithm must be transferred to the appropriate subteam before execution can begin. Thus, the division of processors amongst subteams is also accompanied by a redistribution of data amongst processors. This is a specialization of a general team-parallel model to the case of message-passing machines, since on a shared-memory machine no redistribution would be necessary.

41

The choice of how to balance workload across processors (in this case, choosing the sub-teams such that the time for subsequent data-parallel operations within each team is minimized) while minimizing interprocessor communication (in this case, choosing the subteams such that the minimum time is spent redistributing data) has been proven to be NP-complete [BPS93, KQ95]. Therefore, most realistic systems use heuristics. For divide-and-conquer algorithms, simply maximizing the network locality of processors within subteams is a reasonable choice, even at the cost of increased time to redistribute the subteams. The intuitive reason is that the locality of a team will in turn affect the locality of all future subteams that it creates, and this network locality will affect both the time for subsequent data-parallel operations and the time for redistributing data between future subteams.

3.2.2 Collection-oriented data type

Within a team, all computation is done in a data-parallel fashion, thereby supporting any parallelism present in the divide and merge phases of a divide-and-conquer algorithm. Notionally, this can be thought of as strictly synchronous with all processors executing in lockstep, although in practice this is not required of the implementation. An implementation of the team-parallel programming model must therefore supply a collection-oriented distributed data type [SB91], and a set of data-parallel operations operating on this data type that are capable of expressing the most common forms of divide and merge operations.

Discussion   In this dissertation I have chosen to use vectors (one-dimensional arrays) as the primary distributed data type, since these have well-established semantics and seem to offer the best support for divide-and-conquer algorithms. Axford [Axf92] discusses the use of other collection-oriented data types.

I am also assuming the existence of the following collective communication operations:

Barrier No processor can continue past a barrier until all processors have arrived at it.

Broadcast One processor broadcasts a message of size m to all other processors.

Reduce Given an associative binary operator $\oplus$ and m elements of data on each processor, return to all processors m results. The ith result is computed by combining the ith element on each processor using $\oplus$.

Scan Given an associative binary operator $\otimes$ and m elements of data on each processor, return to all processors m results. The ith result on processor j is computed by combining the ith element from the first $j-1$ processors using $\otimes$. This is also known as the "parallel prefix" operation [Ble87].

42

Gather Given m elements of data on each processor, return to all processors the total of $m \times p$ elements of data.

All-to-all communication Given $m \times p$ elements of data on each processor, exchange m elements with every other processor.

Personalized all-to-all communication Exchange an arbitrary number of elements with every other processor.

All of these operations can be constructed from point-to-point sends and receives [GLDS]. However, their abstraction as high level operations exposes more opportunities for architecture-specific optimizations [MPS$^+$95]. A similar set of primitives have been selected by recent communication libraries, such as MPI [For94] and BSPLib [HMS$^+$97], strengthening the claim that these are a necessary and sufficient set of collective communication operations.

Note that I assume that the scan and reduce operations are present as primitives. In previous models of divide-and-conquer parallelism [Mou90, Axf92], these operations have been used as examples of divide-and-conquer algorithms

3.2.3 Efficient serial code

In the team parallel model, when a team of processors running data-parallel code has recursed down to the point at which it only contains a single processor, it switches to a serial implementation of the algorithm. At this point, all of the parallel constructs reduce to purely local operations. Similarly, the parallel team-splitting operation is replaced by a conventional sequential construct with two (or more) recursive calls. When the recursion has finished, the processors return to the parallel code on their way back up the call tree.

Discussion Using specialized sequential code will be faster than simply running the standard team-parallel code on one processor: even if all parallel calls reduce to null operations (for example, a processor sending a message to itself on a distributed-memory system), we can avoid the overhead of a function call by removing them completely. Two versions of each function are therefore required: one to run on multiple processors using parallel communication functions, and a second specialized to run on a single processor using purely local operations.

This might seem like a minor performance gain, but in the team parallel model we actually expect to spend most of our time in sequential code. For a divide-and-conquer problem of size n on P processors, the expected height of the algorithmic tree is $\log n$, while the expected height of our team recursion tree is $\log P$. Since $n \gg P$, the height of the algorithm tree is much greater than that of the recursion tree. Thus, the bulk of the algorithm will be spent executing serial code on single processors.

The team parallel model also allows the serial version of the divide-and-conquer algorithm to be replaced by user-supplied serial code. This could implement the same algorithm using techniques that have the same complexity but lower constants. For example, serial quicksort could be implemented using the standard technique of moving a pointer in from each end of the data, and exchanging items as necessary. Alternatively, the serial version could use a completely different sorting algorithm that has a lower complexity.

Allowing the programmer to supply specialized serial code in this way implies that the collection-oriented data types must be accessible from serial code with little performance penalty, in order to allow arguments and results to be passed between the serial and parallel code. For the distributed vector type chosen for Machiavelli, this is particularly easy, since on one processor its block distribution reduces to a sequential C array.

3.2.4   Load balancing

For unbalanced problems, the relative sizes of the subteams is chosen to approximate the relative work involved in solving the subproblems. This is a simple form of passive load balancing. However, due to the difference in grain size between the problem size and the machine size, it is at best approximate, and a more aggressive form of load balancing must be provided to deal with imbalances that the team approach cannot handle. The particular form of load balancing is left to the implementation, but must be transparent to the programmer.

Discussion   As a demonstration of why an additional form of load balancing is necessary, consider the following pathological case, where we can show that one processor is left with essentially all of the work.

For simplicity, I will assume that work complexity of the algorithm being implemented is linear, so that the processor teams are being divided according to the sizes of the subproblems. Consider a divide-and-conquer algorithm on P processors and a problem of size n. The worst case of load balancing occurs when the divide stage results in a division of $n - 1$ elements (or units of work), and 1 element. Assuming that team sizes are rounded up, these two subproblems will be assigned to subteams consisting of $P - 1$ processors and 1 processor, respectively. Now assume that the same thing happens for the problem of size $n - 1$ being processed on $P - 1$ processors; it gets divided into subproblems of size $n - 2$ and 1. Taken to its pathological conclusion, in the worst case we have $P - 1$ processors each being assigned 1 unit of work and 1 processor being assigned $n + 1 - P$ units of work (we also have a very unbalanced recursion tree, of depth P instead of the expected $O(\log P)$). For $n \gg P$, this results in one processor being left with essentially of the work.

Of course, if we assume $n \gg P$ then an efficient implementation should not hand one unit of work to one processor, since the costs of transferring the work and receiving the result would

outweight the cost of just doing the work locally. There is a practical minimum problem grain size below which it is not worth subdividing a team and transferring a very small problem to a subteam of one processor. Instead, below this grain size it is faster to process the two recursive calls (one very large, and one very small) serially on one team of processors. However, this optimization only reduces the amount of work that a single processor can end up with by a linear factor. For example, if the smallest grain size happens to be half of the work that a processor would expect to do "on average" (that is, $n/2P$), then by applying the same logic as in the previous paragraph we can see that $P - 1$ processors will each have $n/2P$ data, and one processor will have the remaining $n - (P - 1)(n/2P)$ data, or approximately half of the data.

As noted in Section 3.2.3, most of the algorithm is spent in serial code, and hence load-balancing efforts should be concentrated there. In this dissertation I use a *function shipping* approach to load balancing. This takes advantage of the independence of the recursive calls in a divide-and-conquer algorithm, which allows us to execute one or more of the calls on a different processor, in a manner similar to that of a specialized remote procedure call [Nel81]. As an example, an overloaded processor running serial code in a binary divide-and-conquer algorithm can ship the arguments for one recursive branch of the algorithm to an idle processor, recurse on the second branch, and then wait for the helping processor to return the results of the first branch. If there is more than one idle processor, they can be assigned future recursive calls, either by the original processor or by one of the helping processors. Thus, all of the processors could ultimately be brought into play to load-balance a single function call. The details of determining when a processor is overloaded, and finding an idle processor, depend on the particular implementation—I will describe my approach in Chapter 4.

3.3 Summary

In this chapter I have defined a series of axes along which to classify divide-and-conquer algorithms, for the purpose of considering their parallel implementation. In addition, I have defined the *team parallel* programming model, which is designed to allow the efficient parallel expression of a wider range of divide-and-conquer algorithms than has previously been possible.

Specifically, team parallelism uses *teams* of processors to match the run-time behavior of divide-and-conquer algorithms, and can fully exploit the data parallelism within a team and the control parallelism between them. The teams are matched to the subproblem sizes and run asynchronously with respect to each other. Most computation is done using serial code on individual processors, which eliminates parallel overheads. Active load-balancing is used to cope with any remaining irregular nature of the problem; a function-shipping approach is sufficient because of the independent nature of the recursive calls to a divide-and-conquer algorithm.

The team-parallel programming model can be seen as a practical implementation of the H-PRAM computational model discussed in Chapter 2. Specifically, I am implementing a variant of the private H-PRAM model. It has been further restricted to only allow communication via the collective constructs described in Section 3.2.2, whereas the H-PRAM allows full and arbitrary access to the shared memory of a PRAM corresponding to a team. These restrictions could be removed in a shared-memory version of the team parallel model; I will discuss this in Chapter 8.

Chapter 4

The Machiavelli System

The problem is not parallelizing compilers, but serializing compilers
—Gul Agha, IPPS'96

In this chapter I describe the Machiavelli system, an implementation of the team parallel model for distributed-memory parallel machines. In designing and implementing Machiavelli, my main goal was to have a simple system with which to explore issues involved in the efficient implementation of divide-and-conquer algorithms, and especially those that are unbalanced.

Machiavelli uses vectors as its collection-oriented data structure, and employs MPI [For94] as its parallel communication mechanism. To the user, Machiavelli appears as three main extensions to the C language: the vector data structure and associated functions; a data-parallel construct that allows direct evaluation of vector expressions; and a control-parallel construct that expresses the recursion in a divide-and-conquer algorithm. The data-parallel and control-parallel constructs are translated into C and MPI by a simple preprocessor. Note that this is *not* intended to represent a full or robust language, but rather to be a form of parallel syntactic sugar. As such, it allows algorithms to be expressed in a more concise form, while still allowing access to the underlying C language, MPI communication primitives, and Machiavelli functions.

The remainder of this chapter is arranged as follows. In Section 4.1 I give an overview of the Machiavelli system, and the extensions to C that it implements. The next three sections then describe particular extensions and how they are implemented, namely vectors (Section 4.2), predefined vector functions (Section 4.3), and the data-parallel construct (Section 4.4). There then follow three sections on implementation details that are generally hidden from the programmer: teams (Section 4.5), divide-and-conquer recursion (Section 4.6), and load balancing (Section 4.7). Finally, Section 4.8 summarizes the chapter.

47

4.1 Overview of Machiavelli

Perhaps the simplest way to give an overview of Machiavelli is with an example. Figure 4.1 shows quicksort written in both NESL and Machiavelli.

```
function quicksort(s) =
if (#s < 2) then s
else
  let pivot = s[#s / 2];
       les = {e in s | e < pivot};
       eql = {e in s | e = pivot};
       grt = {e in s | e > pivot};
       sorted = {quicksort(v) : v in [les, grt]}
  in sorted[0] ++ eql ++ sorted[1];


vec_double quicksort (vec_double s)
{
  double pivot;
  vec_double les, eql, grt, left, right, result;

  if (length (s) < 2) {
    return s;
  } else {
    pivot = get (s, length (s) / 2);
    les = {x : x in s | x < pivot};
    eql = {x : x in s | x == pivot};
    grt = {x : x in s | x > pivot};
    free (s);
    split (left = quicksort (les),
           right = quicksort (grt));
    result = append (left, eql, right);
    free (left); free (eql); free (right);
    return result;
  }
}
```

Figure 4.1: Quicksort expressed in NESL (top, from [Ble95]) and Machiavelli (bottom).

The general nature of the two functions is very similar, after allowing for the purely functional style of NESL versus the imperative style of C, and it is easy to see the correspondence between lines in the two listings. In particular, there are the following correspondences:

- The NESL operator #, which returns the length of a vector, correponds to the Machiavelli function length(). Here it is used both to test for the base case, and to select the middle element of the vector for use as a pivot.

- The NESL syntax *vector*[*index*], which extracts an element of a vector, corresponds to the Machiavelli function get (*vector, index*). Here it is used to extract the pivot element.

- The apply-to-each operator { expr: elt in vec | cond } is used in both languages for data-parallel expressions. It is read as "in parallel, for each element *elt* in

48

the vector *vec* such that the condition *cond* holds, return the expression *expr*". Here the apply-to-each operator is being used to select out the elements less than, equal to, and greater than the pivot, to form three new vectors. NESL allows more freedom with this expression: for example, if *expr* is not present it is assumed to be *elt*.

However, there are also significant differences between the two functions:

- The NESL function is polymorphic, and can be applied to any of the numeric types. The Machiavelli function, limited by C's type system, is specialized for a particular type.

- NESL's basic data structure is a vector. In Machiavelli, a vector is specified by prepending the name of a type with `vec_`.

- NESL is garbage collected, whereas in Machiavelli vectors must be explicitly freed.

- Some useful elements of NESL's syntax (such as ++ to append two vectors) have been replaced with function calls in Machiavelli.

- Rather than allowing the application of parallel functions inside an apply-to-all (as on line 8 of the NESL listing), Machiavelli uses an explicit `split` syntax to express control parallelism. This is specialized for the recursion in a divide-and-conquer function.

To produce efficient code from this syntax, the Machiavelli system uses three main components. First, a preprocessor translates the syntactic extensions into pure C and MPI, and produces a parallel and serial version of each user-defined function. Second, a collection of predefined operations is specialized by the preprocessor for the types supplied by the user at compile time (for example, doubles in the above example). Third, a run-time system handles team operations and load balancing. After compiling the generated code with a standard C compiler, it is linked against the run-time library and an MPI library, as shown in Figure 4.2

4.2 Vectors

Machiavelli is built around the *vector* as its basic data structure. A vector is a dynamically-created ordered collection of values, similar to an array in sequential C, but is distributed across a team of processors. After being declared as a C variable, a vector is created when it is first assigned to. It is then only valid within the creating team. Vectors have reference semantics: thus, assigning one vector to another will result in the second vector sharing the same values as the first. To copy the values an explicit data-parallel operation must be used (see Section 4.4). A vector is also strongly typed: it cannot be "cast" to a vector of another type. Again, an explicit data-parallel operation must be used to copy and cast the values into a new vector.
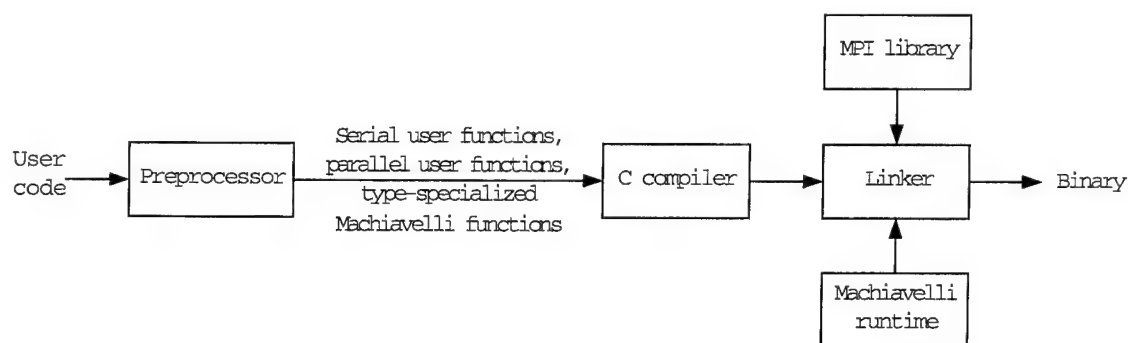
Figure 4.2: Components of the Machiavelli system

Vectors can be created for any of the basic types, and for any types that the user defines. Accessing the fields of vectors of user-defined types is done using the standard C "." operator. For example, given a user-defined type containing floating point fields x and y, and a vector points of such types, a vector of the product of the fields could be computed using:

```
vec_double products = { p.x * p.y :   p in points }
```

4.2.1 Implementation

A vector is represented on every processor within its team by a structure containing its length, the number of elements that are currently on this processor, a pointer to a block of memory containing those elements, and a flag indicating whether the vector is unbalanced or not (this will be discussed in Section 4.6.2).

Machiavelli normally uses a simple block distribution for vectors. This corresponds to the case of a vector balanced across all the processors within a team; the unbalanced case is discussed in Section 4.4.2. Thus, for a vector of size n on P processors, the first processor in the team has the first $\lceil n/P \rceil$ elements, the second processor has the next $\lceil n/P \rceil$ elements, and so on. When the vector size n is not an exact multiple of the number of processors P, the last processor will have fewer than $\lceil n/P \rceil$ elements on it. If $n \leq (P \times P - 3) + 1$ then other trailing processors will also have fewer than $\lceil n/P \rceil$ elements, and in the extreme case of $n = 1$ then $P - 1$ processors will have no elements.

Given this distribution, it is easy to construct efficient functions of the vector length and team size that compute the maximum number of elements per processor (used to allocate space), the number of elements on a given processor, and the processor and offset on that processor that a specific index maps to. This last function is critical for performing irregular data-transfer operations, such as sends or fetches of elements.

50

To allow vectors of user-defined types to be manipulated using MPI operations (for example, to extract an element from such a vector) we must use MPI's derived datatype functionality. This encodes all the relevant information about a C datatype in a single *type descriptor*, which can then be used by MPI communication function to manipulate variables and buffers of the matching type. For every new type used in a vector, the preprocessor therefore generates initialization code to define a matching type descriptor [GLS94]. For example, Figure 4.3 shows a point structure defined by the user, and the function generated by the preprocessor to create the appropriate type descriptor for MPI.

```
/* Structure defined by user */
typedef struct _point {
  double x;
  double y;
  int tag;
} point;

/* Initialization code generated by preprocessor */
MPI_Datatype _mpi_point;

void _mpi_point_init ()
{
  point example;
  int i, count = 2;
  int lengths[2] = { 2, 1 };
  MPI_Aint size, displacements[2];
  MPI_Datatype types[2];

  MPI_Address (&example.x, &displacements[0]);
  types[0] = MPI_DOUBLE;
  MPI_Address (&example.tag, &displacements[1]);
  types[1] = MPI_INT;
  for (i = 1; i >= 0; i--) {
    displs [i] -= displs[0];
  }
  MPI_Type_struct (count, lengths, displacements, types,
                   &_mpi_point);
  MPI_Type_commit (&_mpi_point);
}

/* _mpi_point can now be used as an MPI type */
```

Figure 4.3: MPI type definition code generated for a user-defined type.

4.3   Vector Functions

To operate on the vector types, Machiavelli supplies a range of basic data-parallel vector functions. In choosing which functions to support a trade-off must be made between simplicity (providing a small set of functions that can be implemented easily and efficiently) and generality (providing a larger set of primitives that abstract out more high-level operations). I have

51

chosen to implement a smaller subset of basic operations than are supplied by languages such as NESL. For example, NESL [BHSZ95] provides a high-level `sort` function that in the current implementation is mapped to an equivalent `sort` primitive in the CVL [BCH⁺93] library. No equivalent construct is supplied in the basic Machiavelli system. However, the structure of Machiavelli as a simple preprocessor operating on function templates makes it comparatively easy to add a vector sorting function.

The vector functions can be divided into four basic types: reductions, scans, vector reordering, and vector manipulation.

4.3.1 Reductions

A reduction operation on a vector returns the (scalar) result of combining all elements in the vector using a binary associative operator, $\oplus$. Machiavelli supports the reduction operations shown in Table 4.1. The operations `reduce_min_index` and `reduce_max_index` extend the basic definition of reduction, in that they return the (integer) index of the minimum or maximum element, rather than the element itself.

Function name	Operation	Defined on
`reduce_sum`	Sum	All numeric types
`reduce_product`	Product	All numeric types
`reduce_min`	Minimum value	All numeric types
`reduce_max`	Maximum value	All numeric types
`reduce_min_index`	Index of minimum	All numeric types
`reduce_max_index`	Index of maximum	All numeric types
`reduce_and`	Logical and	Integer types
`reduce_or`	Logical or	Integer types
`reduce_xor`	Logical exclusive-or	Integer types

Table 4.1: The reduction operations supported by Machiavelli. "All numeric types" refers to C's short, int, long, float, and double types. "Integer types" refers to C's short, int, and long types.

The implementation of reduction operations is very straightforward. Every processor performs a loop over its own section of the vector, accumulating results into a variable. They then combine the accumulated local results in an `MPI_Allreduce` operation, which returns a global result to all of the processors. The preprocessor generates reduction functions specialized for a particular type and operation as necessary. As an example, Figure 4.4 shows the parallel implementation of `reduce_min_double`, which returns the minimum element of a vector of doubles. The use of the team structure passed in the first argument will be explained in Section 4.5.

52

```
double reduce_min_double (team *tm, vec_double src)
{
  double global, local = DBL_MAX;
  int i, nelt = src.nelt_here;

  for (i = 0; i < nelt; i++) {
    double x = src.data[i];
    if (x < local) local = x;
  }
  MPI_Allreduce (&local, &global, 1, MPI_DOUBLE, MPI_MIN, tm->com);
  return global;
}
```

Figure 4.4: Parallel implementation of reduce_min for doubles, a specialization of the basic reduction template for summing the elements of a vector of doubles.

4.3.2 Scans

A scan, or parallel prefix, operation can be thought of as a generalized reduction. Take a vector v of length n, containing elements v_0, v_1, v_2, ..., and an associative binary operator $\otimes$ with an identity value of $\mathrm{id}_\otimes$. A scan of v returns a vector of the same length n, where the element v_i has the value $\mathrm{id}_\otimes \otimes v_0 \otimes v_1 \otimes \ldots \otimes v_{i-1}$. Note that this is the "exclusive" scan operation; the inclusive scan operation does not use the identity value, and instead sets the value of v_i to $v_0 \otimes v_1 \otimes \ldots \otimes v_i$. Machiavelli supplies a slightly smaller range of scans than of reductions, as shown in Table 4.2, because there is no equivalent of the maximum and minimum index operations.

Function name	Operation	Defined on
scan_sum	Sum	All numeric types
scan_product	Product	All numeric types
scan_min	Minimum value	All numeric types
scan_max	Maximum value	All numeric types
scan_and	Logical and	Integer types
scan_or	Logical or	Integer types
scan_xor	Logical exclusive-or	Integer types

Table 4.2: The scan operations supported by Machiavelli. Types are as in Table 4.1.

Scans are only slightly more difficult to implement than reductions. Again, every processor performs a loop over its own section of the vector, accumulating a local result. The processors then combine their local results using an MPI_Scan operation, which returns an intermediate scan value to each processor. A second local loop is then performed, combining this scan value with the original source values in the vector to create the result vector. There is an additional complication in that MPI provides an inclusive scan rather than an exclusive one. For

53

operations with a computable inverse (for example, sum) the exclusive scan can be computed by applying the inverse operation to the inclusive result and the local intermediate result. For operations without a computable inverse (for example, min), the inclusive scan is computed and the results are then shifted one processor to the "right" using MPI_Sendrecv. As an example, Figure 4.5 shows the parallel implementation of scan_sum_int, which returns the exclusive prefix sum of a vector of integers.

```
vec_int scan_sum_int (team *tm, vec_int src)
{
  int i, nelt = src.nelt_here;
  int incl, excl, swap, local = 0;
  vec_int result;

  result = alloc_vec_int (tm, src.length);

  /* Local serial exclusive scan */
  for (i = 0; i < nelt; i++) {
    swap = local;
    local += src.data[i];
    dst.data[i] = swap;
  }

  /* Turn inclusive MPI scan into exclusive result */
  MPI_Scan (&local, &incl, 1, MPI_INT, MPI_SUM, tm->com);
  excl = incl - local;

  /* Combine exclusive result with previous scan */
  for (i = 0; i < nelt; i++) {
    dst.data[i] += excl;
  }
  return result;
}
```

Figure 4.5: Parallel implementation of scan_sum for integers, a specialization of the basic scan template for performing a prefix sum of a vector of integers. Note the correction from the inclusive result of the MPI function to obtain an exclusive scan.

4.3.3 Vector reordering

There are two basic vector reordering functions, send and fetch, which transfer source elements to a destination vector according to an index vector. In addition, the function pack, which is used to redistribute the data in an unbalanced vector (see Section 4.4.2) can be seen as a specialized form of the send function. These are the most complicated Machiavelli functions, but they can be implemented using only one call to MPI_Alltoall to set up the communication, and one call to MPI_Alltoallv to actually perform the data transfer.

send (*vec_source, vec_indices, vec_dest*)

send is an indexed vector write operation. It sends the values from the source vector

54

vec_source to the positions specified by the index vector *vec_indices* in the destination vector *vec_dest*, so that *vec_dest[vec_indices[i]] = vec_source[i]*.

This is implemented using the following operations on each processor. For simplicity, I assume that there are P processors, that the vectors are of length n, and that there are exactly n/P elements on each processor.

1. Create two arrays, *num_to_send[P]* and *num_to_recv[P]*. These will be used to store the number of elements to be sent to and received from every other processor, respectively.

2. Iterate over this processor's n/P local elements of *vec_indices*. For every index element i, compute the processor q that it maps to, and increment *num_to_send[q]*. Each processor now knows how many elements it will send to every other processor.

3. Exchange *num_to_send[P]* with every other processor using MPI_Alltoall(). The result of this is *num_to_recv[P]*, the number of elements to receive from every other processor.

4. Allocate a data array *data_to_send[n/P]* and an index array *indices_to_send[n/P]*. These will be used to buffer data and indices before sending to other processors. Similarly, allocate a data array *data_to_recv[n/P]* and an index array *indices_to_recv[n/P]*. Notionally the data and index arrays are allocated and indexed separately, although in practice they can be allocated as an array of structures to improve locality.

5. Perform a plus-scan over *num_to_recv[]* and *num_to_send[]*, resulting in arrays of offsets *send_ptr[P]* and *recv_ptr[P]*. These offsets will act as pointers into the *data_* and *indices_* arrays.

6. Iterate over this processor's n/P local elements of *vec_indices[]*. For each element *vec_indices[i]*, compute the processor q and offset o that it maps to. Fetch and increment the current pointer, *ptr = send_ptr[q]++*. Copy *vec_source[i]* to *data_to_send[ptr]*, and o to *indices_to_send[ptr]*.

7. Call MPI_Alltoallv(), sending data from *data_to_send[]* according to the element counts in *num_to_send[]*, and receiving into *data_to_recv[]* according to the counts in *num_to_recv[]*. Do the same for *indices_to_send[]*.

8. Iterate over *data_to_recv[]* and *indices_to_recv[]*, performing the indexed write operation *vec_dest[indices_to_recv[i]] = data_to_recv[i]*.

Note that steps 1–3 and 5 are independent of the particular data type being sent. They are therefore abstracted out into library functions. The remaining steps are type-dependent, and are generated as a function by the preprocessor for every type that is the subject of a send.

55

fetch (*vec_source*, *vec_indices*, *vec_dest*)

fetch is an indexed vector read operation. It fetches data values from the source vector *vec_source* (from the positions specified by the index vector *vec_indices*) and stores them in the destination vector *vec_dest*, so that *vec_dest[i]* = *vec_source[vec_indices[i]]*.

Obviously, this could be implemented using two send operations—one to transfer the indices of the requested data items to the processors that hold the data, and a second to transfer the data back to the requesting processors. However, by combining them into a single function some redundant actions can be removed, since we know ahead of time how many items to send and receive in the second transfer. Again, for simplicity I assume that there are P processors, that all the vectors are of length n, and that there are exactly n/P elements on each processor.

1. Create two arrays, *num_to_request[P]* and *num_to_recv[P]*. These will be used to store the number of requests to be sent to every other processor, and the number of requests to be received by every other processor, respectively.

2. Iterate over this processor's n/P local elements of *vec_indices*. For every index element i, compute the processor q that it maps to, and increment *num_to_request[q]*. Each processor now knows how many elements it will request from every other processor.

3. Exchange *num_to_request[]* with every other processor using MPI_Alltoall(). The result of this is *num_to_recv[]*, the number of requests to receive from every other processor (which is the same as the number of elements to send).

4. Allocate an index array *indices_to_request[n/P]*. This will be used to buffer indices to request before sending to other processors. Similarly, allocate an index array *indices_to_recv[n/P]*.

5. Perform a plus-scan over *num_to_request[]* and *num_to_recv[]*, resulting in arrays of offsets *request_ptr[P]* and *recv_ptr[P]*. These offsets will act as pointers into the *indices_to_request[]* and *indices_to_recv[]* arrays.

6. Allocate an index array *requested_index[n/P]*. This will store the index in a received data buffer that we will eventually fetch the data from.

7. Iterate over this processor's n/P local elements of *vec_indices[]*. For each element *vec_indices[i]*, compute the processor q and offset o that it maps to. Fetch and increment the current pointer, *ptr = request_ptr[q]++*. Copy o to *indices_to_request[ptr]*.

8. Call MPI_Alltoallv(), sending data from *request_to_send[]* according to the element counts in *num_to_request[]*, and receiving into *request_to_recv[]* according to the element counts in *num_to_recv[]*.

9. Allocate data arrays *data_to_send[n/P]* and *data_to_recv[n/P]*.

10. Iterate over *request_to_recv[]*, extracting each offset in turn, fetching the requested element, and storing it in the data buffer, *data_to_send[i]* = *vec_dest[request_to_recv[i]]*.

11. Call `MPI_Alltoallv()`, sending data from *data_to_send[]* according to the element counts in *num_to_recv[]*, and receiving into *data_to_recv[]* according to the element counts in *num_to_request[]*.

12. Iterate over *data_to_recv[]* and *requested_index[]*, performing the indexed write operation *vec_dest[i]* = *data_to_recv[requested_index[i]]*.

Again, steps 1–8 are independent of the particular data type being requested, and abstracted out into library functions, while the remaining steps are generated as a function by the preprocessor for every type that is the subject of a `fetch`.

pack (*vec_source*)

`pack` redistributes the data in an unbalanced vector so that it has the block distribution property described in Section 4.2. An unbalanced vector (that is, one with an arbitrary amount of data on each processor) can be formed using an apply-to-each operator with a conditional (see Section 4.4) or a lazy append on the results of recursive function calls (see Section 4.5). The `pack` function is normally called as part of another Machiavelli operation.

`pack` is simpler than `send` since we can transfer contiguous blocks of elements, rather than sequences of elements with the appropriate offsets to store them in.

1. Exchange *vec_source.nelt_here* with every other processor using `MPI_Alltoall()`. The result of this is *num_on_each[P]*.

2. Perform a plus-scan across *num_on_each[P]* into *first_on_each[P]*. The final result of the plus scan is the total length *n* of the vector.

3. From *n*, compute the number of elements per processor in the final block distribution, *final_on_each[P]*, and allocate a receiving array *data_to_recv[n/P]*.

4. Allocate two arrays, *num_to_recv[P]* and *num_to_send[P]*.

5. Iterate over *final_on_each[P]*, computing which processor(s) will contribute data for each destination processor in turn. If this processor will be receiving, update *num_to_recv[]*. If this processor will be sending, update *num_to_send[]*.

6. Call `MPI_Alltoallv()`, sending data from *vec_source.data[]* according to the element counts in *num_to_send[]*, and receiving into *data_to_recv[]* according to the element counts in *num_to_recv[]*.

7. Free the old data storage in *vec_source.data[]* and replace it with *data_to_recv[]*.

57

4.3.4 Vector manipulation

Machiavelli also supplies seven functions that manipulate vectors in various ways. Most of these have very simple implementations. All but `length` and `free` are specialized for the particular type of vector being manipulated.

free (*vector*)

Frees the memory associated with vector *vector*.

new_vec (*n*)

Returns a vector of length *n*. This is translated in the parallel and serial code to calls to the Machiavelli functions `alloc_vec_type` and `alloc_vec_type_serial`, respectively.

length (*vector*)

Returns the length field of the vector structure.

get (*vector*, *index*)

Returns the value of the element of vector *vector* at index *index*. Using the length of *vector*, every processor computes the processor and offset that *index* maps to. The processors then perform a collective `MPI_Broadcast` operation, where the processor that holds the value contributes the result. As an example, Figure 4.6 shows the parallel implementation of `get` for a user-defined `point` type.

```
point get_point (team *tm, vec_point src, int i)
{
  point result;
  int proc, offset;

  proc_and_offset (i, src.length, tm->nproc, &proc, &offset);
  if (proc == tm->rank) {
    dst = src.data[offset];
  }
  MPI_Bcast (&result, 1, _mpi_point, proc, tm->com);
  return result;
}
```

Figure 4.6: Parallel implementation of `get` for a user-defined `point` type

set (*vector*, *index*, *value*)

Sets the element at index *index* of vector *vector* to the value *value*. Again, every processor computes the processor and offset that *index* maps to. The processor that holds the element then sets its value. As an example, Figure 4.7 shows the parallel implementation of `set` for a vector of characters.

58

```
void set_char (team *tm, vec_char dst, int i, char elt)
{
  int proc, offset;

  proc_and_offset (i, src.length, tm->nproc, &proc, &offset);
  if (proc == tm->rank) {
    dst.data[offset] = elt;
  }
}
```

Figure 4.7: Parallel implementation of `set` for vectors of characters

index (*length, start, increment*)

Returns a vector of length *length*, containing the numbers *start, start + increment, start + 2*increment, ...* This is implemented as a purely local loop on each processor, and is specialized for each numeric type. As an example, Figure 4.8 shows the parallel implementation of `index` for integer vectors.

```
vec_int index_int (team *tm, int len, int start, int incr)
{
  int i, nelt;
  vec_int result;

  /* Start counting from first element on this processor */
  start += first_elt_on_proc (tm->this_proc) * incr;
  result = alloc_vec_int (tm, len);
  nelt = result.nelt_here;

  for (i = 0; i < nelt; i++, start += incr)
    result.data[i] = start;
  }
  return result;
}
```

Figure 4.8: Parallel implementation of `index` for integer vectors

distribute (*length, value*)

Returns a vector of length *length*, containing the value *value* in each element. This is defined for any user-defined type, as well as for the basic C types. Again, it is implemented with a purely local loop on each processor, as shown in Figure 4.9.

vector (*scalar*)

Returns a single-element vector containing the variable *scalar*. This is equivalent to `dist` (1, *scalar*), and is provided merely as a convenient shorthand.

replicate (*vector, n*)

Given a vector of length m, and an integer n, returns a vector of length m^n, containing n copies of *vector*. This is converted into a doubly-nested loop in serial code (see Figure 4.10), and into a sequence of n send operations in parallel code.

```
vec_double distribute_double (team *tm, int len, double elt)
{
  int i, nelt;
  vec_int result;

  result = alloc_vec_double (tm, len);
  nelt = result.nelt_here;

  for (i = 0; i < nelt; i++)
    result.data[i] = elt;
  }
  return result;
}
```

Figure 4.9: Parallel implementation of distribute for double vectors

```
vec_pair replicate_vec_pair_serial (vec_int src, int n)
{
  int i, j, r, nelt;
  vec_pair result;

  nelt = src->nelt_here;
  result = alloc_vec_pair_serial (nelt * n);
  r = 0;

  for (i = 0; i < n; i++) {
    for (j = 0; j < nelt; j++) {
      result.data[r++] = src->data[j];
    }
  }
  return result;
}
```

Figure 4.10: Serial implementation of replicate for a user-defined pair type

append (*vector, vector [, vector]*)

Appends two or more vectors together, returning the result of their concatenation as a new vector. This is implemented as successive calls to a variant of the pack function (see Section 4.4). Here it is used to redistribute the elements of a vector which is spread equally amongst the processors to a smaller subset of processors, representing a particular section of a longer vector. The Machiavelli preprocessor converts an n-argument append to n successive calls to pack, each to a different portion of the result vector. As an example, Figure 4.11 shows the implementation of append for three integer vectors.

The next four functions (odd, even, interleave, and transpose) can all be constructed from send combined with other primitives. However, providing direct functions allows for a more efficient implementation by removing the need for the use of generalised indexing. That is, each of the four functions preserves some property in its result that allows us to precompute the addresses to send blocks of elements to, rather than being forced to compute the address for every element, as in send.

60

```
vec_int append_3_vec_int (team *tm, vec_int vec_1,
                          vec_int vec_2, vec_int vec_3)
{
    int len_1 = vec_1.length;
    int len_2 = vec_2.length;
    int len_3 = vec_3.length;
    vec_int result = alloc_vec_int (tm, len_1 + len_2 + len_3);

    pack_vec_int (tm, result, vec_1, 0);
    pack_vec_int (tm, result, vec_2, len_1);
    pack_vec_int (tm, result, vec_3, len_1 + len_2);
}
```

Figure 4.11: Parallel implementation of `append` for three integer vectors

even (*vector*, *n*)
odd (*vector*, *n*)

Given a vector *vector*, and an integer *n*, `even` returns the vector composed of the even-numbered blocks of elements of length *n* from *vector*. Thus, `even (foo, 3)` returns the elements $0, 1, 2. 6. 7. 8. 12. 13. 14, \ldots$ of vector `foo`. `odd` does the same, but for the odd-numbered blocks of elements. The length of *vector* is assumed to be an exact multiple of twice the block-size *n*. As an example, Figure 4.12 shows the serial implementation of `even` for a user-defined pair type. The parallel implementations of `odd` and `even` simply discard the even and odd elements respectively, returning an unbalanced vector. Note that the use of generalised odd and even primitives (rather than just single-element odd and even) allows them to be used for other purposes. For example, `even(bar,length(bar)/2)` returns the first half of vector `bar`.

```
vec_pair even_vec_pair (vec_pair *src, int blocksize)
{
    int i, j, r, nelt;
    vec_pair result;

    nelt = src->nelt_here;
    alloc_vec_pair (nelt / 2, &result);
    r = 0;

    for (i = 0; i < nelt; i += blocksize) {
        for (j = 0; j < blocksize; j++) {
            result.data[r++] = src->data[i++];
        }
    }
    return result;
}
```

Figure 4.12: Serial implementation of `even` for a vector of user-defined pairs.

interleave (*vec1*, *vec2*, *n*)

Given two vectors *vec1* and *vec2*, and an integer *n*, returns the vector composed of the first *n* elements from *vec1*, followed by the first *n* elements from *vec2*, followed by the second *n*

61

elements from *vec1*, and so on. As such, it does the opposite of even and odd. The lengths of *vec1* and *vec2* are assumed to be the same, and an exact multiple of the blocksize *n*. Again, the use of a generalised interleave primitive allows it to be used for other purposes. For example, given two $m \times n$ matrices A and B, interleave (A, B, n) returns the $2m \times n$ matrix whose rows consist of the appended rows of A and B.

transpose (*vector, m, n*)

Given a vector *vector* which represents an $m \times n$ matrix, returns the vector representing the transposed $n \times m$ matrix.

4.4 Data-Parallel Operations

For general data-parallel computation, Machiavelli uses the apply-to-each operator, which has the following syntax:

```
{ expr:  elt in vec [, elt in vec] [| cond] }
```

expr is any expression (without side-effects) that can be the right-hand side of an assignment in C. *elt* is an iteration variable over a vector *vec*. The iteration variable is local to the body of the apply-to-all. There may be more than one vector, but they must have the same length. *cond* is any expression without side-effects that can be a conditional in C.

The effect of this construct is to iterate over the source vector(s), testing whether the condition is true for each element, and if so evaluating the expression and writing the result to the destination vector.

4.4.1 Implementation

The Machiavelli preprocessor converts an apply-to-each operation without a conditional into a purely local loop on each processor, iterating over the source vectors and writing the resultant expression for each element into the destination vector. As an example, Figure 4.13 shows a simple data-parallel operation and the resulting code.

The absence of synchronization between processors explains why the expressions within an apply-to-each cannot rely on side effects; any such effects would be per-processor, rather than global across the machine. In general, this means that C's pre- and post- operations to increment and decrement variables cannot be used inside an apply-to-each. The independence of loop expressions also enables the Machiavelli preprocessor to perform loop fusion on adjacent apply-to-each operations that are iterating across the same vectors.

```
/* Machiavelli code generated from:
 *    vec_double diffs = {(elt - x_mean)^2 : elt in x}
 */
{
  int i, nelt = x.nelt_here;

  diffs = alloc_vec_double (tm, x.length);

  for (i = 0; i < nelt; i++) {
    double elt = x.data[i];
    diffs.data[i] = (elt - x_xmean)^2;
  }
}
```

Figure 4.13: Parallel implementation of an apply-to-each operation

4.4.2 Unbalanced vectors

If a conditional is used in an apply-to-each, then the per-processor pieces of the destination vector may not have the same length as the pieces of source vector(s). The result is that we are left with an unbalanced vector; that is, one in which the amount of data per processor is not fixed. This is marked as such using an "unbalanced" flag in its vector structure. As an example, Figure 4.14 shows the parallel implementation of an apply-to-each with a simple conditional.

```
/* Machiavelli code generated from:
 *    vec_double result;
 *    result = {(val - mean)^2 : val in values, flag in flags
 *                               | (flag != 0)};
 */
{
  int i, ntrue = 0, nelt = values.nelt_here;

  /* Overallocate the result vector */
  result = alloc_vec_double (tm, values.length);

  /* ntrue counts conditionals */
  for (i = 0; i < nelt; i++) {
    int flag = flags.data[i];
    if (flags != 0) {
      double val = values.data[i];
      result.data[ntrue++] = (val - xmean)^2;
    }
  }
  /* Mark the result as unbalanced */
  result.nelt_here = ntrue;
  result.unbalanced = true;
}
```

Figure 4.14: Parallel implementation of an apply-to-each with a conditional

An unbalanced vector can be balanced (that is, its data can be evenly redistributed across the processors) using a `pack` function, as described in Section 4.3.3. The advantage of *not* balancing a vector is that by not calling `pack` we avoid two MPI collective operations, one of which

63

transfers a small and fixed amount of information between processors (MPI_Allgather) while the other may transfer a large and varying amount of data (MPI_Alltoallv).

Given an unbalanced vector, we can still perform many operations on it in its unbalanced state. In particular, reductions, scans, and apply-to-each operations (including those with conditionals) that operate on a single vector are all oblivious to whether their input vector is balanced or unbalanced, since they merely loop over the number of elements present on each processor. Given the underlying assumption that local operations are much cheaper than transferring data between processors, it is likely that the time saved by avoiding data movement in this way outweighs the time lost in subsequent operations caused by not balancing the data across the processors, and hence resulting in all other processors waiting for the processor with the most data. As discussed in Chapter 2, Sipelstein is investigating the tradeoffs and issues involved in automating the decision about when to perform a pack operation so as to minimize the overall running time. Machiavelli only performs a pack when required, but allows the user to manually insert additional pack operations.

The remaining Machiavelli operations that operate on vectors all require their input vectors to be packed before they can proceed. The implementations of get, set, send, fetch are therefore extended with a simple conditional that tests the "unbalanced" flag of their input vector structures, and performs a pack on any that are unbalanced. These operations share the common need to quickly compute the processor and offset that a specific vector index maps to; the balanced block distribution satisfies this requirement. Apply-to-each operations on multiple vectors are also extended with test-and-pack, although in this case the requirement is to assure that vectors being iterated across in the same loop share the same distribution.

Note that an unbalanced vector in the team parallel model is equivalent to a standard vector in the concatenated model described in Chapter 3, in that it only supports those operations that do not require rapid lookup of element locations. Two further optimizations that are possible when using unbalanced vectors are discussed in the next section.

4.5 Teams

Machiavelli uses *teams* to express control-parallel behavior between data-parallel sections of code, and in particular to represent the recursive branches of a divide-and-conquer algorithm. A team is a collection of processors, and acts as the context for all functions on vectors within it. A vector is distributed across the processors of its owning team, and can only be operated on by data-parallel functions within that team. Teams are divided when a divide-and-conquer algorithm makes recursive calls, and merged when the code returns from the calls. Otherwise, teams are mutually independent, and do not communicate or synchronize with each other. However, unless the programmer wants to bypass the preprocessor and gain direct access to the underlying team functions, the existence of teams is effectively hidden.

64

4.5.1 Implementation

A team is represented by the MPI concept of a communicator, which encapsulates most of what we want in a team. Specifically, a communicator describes a specific collection of processors, and when passed to an MPI communication function restricts the "visible universe" of that communication function to the processors present in the communicator.

The internal representation of a team consists of a structure containing the MPI communicator, the number of processors in the team, and the rank of this processor in the team. All processors begin in a "global" team, which is then subdivided by divide-and-conquer algorithms to form smaller teams. The preprocessor adds a pointer to the current team structure as an extra argument to every parallel function, as was seen in the implementations of Machiavelli functions in Section 4.3. In this way, the subdivision of teams on the way down a recursion tree, and their merging together on the way up the tree, is naturally encoded in the passing of smaller teams as arguments to recursive calls, and reverting to the parent teams when returning from a call.

4.6 Divide-and-conquer Recursion

Machiavelli uses the syntax

$$\texttt{split}\ (result_1 = func\ (arg_1),\ result_2 = func\ (arg_2)\ [,\ result_n = func\ (arg_n)])$$

to represent the act of performing divide-and-conquer function calls. var_n is the result returned by invoking the function $func$ on the argument list arg_n. In team parallelism, this is implemented by dividing the current team into one subteam per function call, sending the arguments to the subteams, recursing, and then fetching the results from the subteams. Each of these steps will be described. Note that $func$ must be the same for every call in a given $\texttt{split}$.

4.6.1 Computing team sizes

Before subdividing a team into two or more subteams, we need to know how many processors to allocate to each team. For Machiavelli's approximate load balancing of teams, the subteam sizes should be chosen according to the relative amount of expected work that the subtasks are expected to require. This is computed at runtime by calling an auxiliary cost function defined for the divide-and-conquer function. The cost function takes the same arguments as the divide-and-conquer function, but returns as a result an integer representing the relative amount of work that those arguments will require. By default, the preprocessor generates a cost function that returns the size of the first vector in the argument list (that is, it assumes that the cost will be

a linear function of the first vector argument). This can be overridden for a particular divide-and-conquer function `divconq` by defining a cost function `divconcq_cost`. As an example, Figure 4.15 shows a simple cost function for quicksort. The results of a cost function have no units, since they are merely compared to each other. The actual act of subdividing a team is performed with the `MPI_Comm_split` function that, when given a flag declaring which new subteam this processor should join, creates the appropriate MPI communicator.

```
int quicksort_cost (vec_double s) {
  int n = length (s);
  return (n * (int) log ((double) n));
}
```

Figure 4.15: Machiavelli cost function for quicksort

4.6.2   Transferring arguments and results

Having created the subteams, we must redistribute any vector arguments to the respective subteams (scalar arguments are already available on each processor, since we are programming in an SPMD style). The task is to transfer each vector to a smaller subset of processors. This can be accomplished with a specialized form of the `pack` function; all that is necessary is to change the computation of the number of elements per processor for the destination vector, and to supply a "processor offset" that serves as the starting point for the subset of processors. However, there are two optimizations that can be made to reduce the number of collective operations.

First, the redistribution function can accept unbalanced vectors as arguments, just as the `pack` function can. This is particularly important for divide-and-conquer functions, where the arguments to recursive calls may be computed using a conditional in an apply-to-each, which results in unbalanced vectors. Without the ability to redistribute these unbalanced vectors, the number of collective communication steps would be doubled (first to balance the vector across the original team, and then to redistribute the balanced vector to a smaller subteam).

Second, the redistribution function can use a single call to `MPI_Alltoallv()` to redistribute a vector to each of the subteams. Consider the recursion in quicksort:

```
split (left = quicksort (les),
       right = quicksort (grt));
```

les and *grt* are being supplied as the argument to recursive function calls that will take place in different subteams. Since these subteams are disjoint, a given processor will send data from either *les* or *grt* to any other given processor, but never from both. We can therefore give `MPI_Alltoallv` the appropriate pointer for the data to send to each of the other processors, sending from *les* to processors in the first subteam and from *grt* to processors in the second subteam. Thus, only one call to `MPI_Alltoallv` is needed for each of the vector arguments to a function.

66

After the recursive function call, we merge teams again, and must now extract a result vector from each subteam, and redistribute it across the original (and larger) team. This can be accomplished by simply setting the "unbalanced" flag of each result vector, and relying on later operations that need the vector in a balanced form to redistribute it across the new, larger team. This can be seen as a reverse form of the "don't pack until we recurse" optimization that was outlined above—now, we don't *expand* a vector until we need to.

4.6.3 Serial code

The Machiavelli preprocessor generates two versions of each user-defined and inbuilt function. The first version, as described in previous sections, uses MPI in parallel constructs, and team-parallelism in recursive calls. The second version is specialized for single processors with purely local data (that is, a team size of one). In this version, apply-to-each constructs reduce to simple loops, as do the predefined vector operations, and the team-based recursion is replaced with simple recursive calls. As was previously discussed, this results in much more efficient code. For example, Figure 4.16 shows the serial implementation of a `fetch` function, which can be compared to the twelve-step description of the parallel equivalent in Section 4.3.3

```
void fetch_vec_int_serial (vec_int src, vec_int indices, vec_int dst)
{
  int i, nelt = dst.nelt_here;

  for (i = 0; i < nelt; i++) {
    dst[i] = src[indices[i]]
  }
}
```

Figure 4.16: Serial implementation of `fetch` for integers

Where more efficient serial algorithms are available, they can be used in place of the serial versions of parallel algorithms that Machiavelli compiles. Specifically, the user can force the default serial version of a parallel function to be overridden by defining a function whose name matches that of the parallel function but has the added suffix "_serial". For example, Figure 4.17 shows a more efficient serial implementation of quicksort supplied by the user:

MPI functions can also be used within Machiavelli code, in cases where the primitives provided are too restrictive. As explained in Section 4.2.1, all user-defined types have equivalent MPI types constructed by the Machiavelli preprocessor, and these MPI types can be used to transfer instances of those types in MPI calls. The fields of a vector and of the current team structure (always available as the pointer `tm` in parallel code) are also accessible, as shown in Figure 4.18.

Putting all these steps together, the code generated by the preprocessor for the recursion in the quicksort from Figure 4.1 is shown in Figure 4.19.

67

```
void user_quicksort (double *A, int p, int r)
{
  if (p < r) {
    double x = A[p];
    int i = p - 1;
    int j = r + 1;
    while (1) {
      do { j--; } while (A[j] > x);
      do { i++; } while (A[i] < x);
      if (i < j) {
        double swap = A[i];
        A[i] = A[j];
        A[j] = swap;
      } else {
        break;
      }
    }
    user_quicksort (A, p, j);
    user_quicksort (A, j+1, r);
  }
}

vec_int quicksort_serial (vec_int src)
{
  user_quicksort (src.data, 0, src.length - 1);
  return src;
}
```

Figure 4.17: User-supplied serial code for quicksort

4.7 Load Balancing

As mentioned in Chapter 3, the team-parallel model restricts load balancing to individual processors because that is where most of the time is spent when $n \gg P$. We are thus trying to cope with the situation where some processors finish first and are idle. Our goal is to ship function invocations from working processors to idle processors, and then return the result to the originating processor.

```
typedef struct _vector {
  void *data;              /* pointer to elements on this processor */
  int nelt_here;           /* number of elements on this processor */
  int length;              /* length of the vector */
} vector;

typedef struct _team {
  int nproc;               /* number of processors in this team */
  int rank;                /* rank of this processor within team */
  MPI_Communicator com;    /* MPI communicator containing team
} team;
```

Figure 4.18: User-accessible fields of vector and team structures

68

```
/* Machiavelli code generated from:
 *   split (left = quicksort (les),
 *          right = quicksort (grt)); */

/* Compute the costs of the two recursive calls */
cost_0 = quicksort_cost (tm, les);
cost_1 = quicksort_cost (tm, grt);

/* Calculate the team sizes, and which subteam I'll join */
which_team = calc_2_team_sizes (tm, cost_0, cost_1, &size_0, &size_1);

/* Create a new team */
create_new_teams (tm, which_team, size_0, &new_team);

/* Allocate an argument vector to hold the result in the new team */
arg = alloc_vec_double (&new_team, which_team ? grt.length : les.length);

/* Compute communication patterns for pack, store in global arrays */
setup_pack_to_two_subteams (tm, les.nelt_here, grt.nelt_here, grt - les,
                            les.length, grt.length, size_0, size_1);

/* Perform the actual data movement of pack, into the argument vector */
MPI_Alltoallv (les.data, _send_count, _send_disp, MPI_DOUBLE,
               arg.data, _recv_count, _recv_disp, MPI_DOUBLE, tm->com);

/* We now perform the recursion in two different subteams */
if (new_team.nproc == 1) {
  /* If new team size is 1, run serial quicksort on the argument */
  result = quicksort_serial (arg);
} else {
  /* Else recurse in the new team */
  result = quicksort (&new_team, arg);
}

/* Returning from the recursion, we rejoin original team, discard old */
free_team (&new_team);

/* Assign tmp to left and nullify right, or vice versa, forming two
 * unbalanced vectors, each on a subset of the processors in the team */
if (which_team == 1) {
  right = tmp;
  left.nelt_here = 0;
} else {
  left = tmp;
  right.nelt_here = 0;
}
```

Figure 4.19: Divide-and-conquer recursion in Machiavelli, showing the user code and the resulting code generated by the preprocessor

69

4.7.1 Basic concept

The implementation of Machiavelli's load-balancing system is restricted by the capabilities of MPI, and by our desire to include as few parallel constructs as possible in the execution of sequential code on single processors. In the absence of threads or the ability to interrupt another processor, an idle processor is unable steal work from another processor [BJK+95], because the "victim" would have to be involved in receiving the steal message and sending the data. Therefore, the working processors must request help from idle processors. The request can't use a broadcast operation because MPI broadcasts are collective operations that all processors must be involved in, and therefore other working processors would block the progress of a request for help. Similarly, the working processor can't use a randomized algorithm to ask one of its neighbors for help (in the hope that it picks an idle processor), because the neighbor might also be working, which would cause the request would block. The only way that we can know that the processor we're requesting help from is idle (and hence can respond quickly to our request) is if we dedicate one processor to keeping track of each processor's status—a manager.

Given this constraint, the load-balancing process takes place as follows. The preprocessor inserts a load-balance test into the sequential version of every divide-and-conquer function. This test determines whether to ask the manager for help with one or more of the recursive function calls. If so, a short message is sent to the manager, containing the total size of the arguments to the function call. Otherwise, the processor continues with sequential execution of the function.

For binary divide-and-conquer algorithms, the requesting processor then blocks waiting for a reply. There is no point in postponing the test for a reply until after the requesting processor has finished the second recursive call, since at that point it would be faster to process the first recursive call locally than to send the arguments to a remote processor, wait for it to finish, and then receive the results. For divide-and-conquer algorithms with a branching factor greater than two, the requesting processor can proceed with another recursive call while waiting for the reply.

The manager sits in a message-driven loop, maintaining a list of idle processors. If it knows of no idle processors when it receives a request, it responds with a "no help available" message. Otherwise, it removes an idle processor from its list, and sends the idle processor a message instructing it to help the requesting processor with a problem of the reported size.

The idle processor has been similarly sitting in a message-driven loop, which it entered after finishing the sequential phase of its divide-and-conquer algorithm. On receiving the message from the manager it sets up appropriate receive buffers for the arguments, and then sends the requesting processor an acknowledgement message signalling its readiness. This three-way handshaking protocol lets us guarantee that the receiver has allocated buffers before the arguments are sent, which can result in faster performance from some MPI implementations

70

by avoiding system buffering. It also avoids possible race conditions in the communication between the three processors.

If the requesting processor receives a "no help available" message from the master, it continues with sequential execution. Otherwise, it receives an acknowledgement message from an idle processor, at which point it sends the arguments, continues with the second recursive function call, and then waits to receive the result of the first function call.

The idle processor receives the function arguments, invokes the function on them, and sends the result back to the requesting processor. It then notifies the manager that it is once again idle, and waits for another message. When all processors are idle, the manager sends every processor a message that causes it to exit its message-driven loop and return to parallel code. A sequence of load-balancing events taking place between two worker processors and one manager is shown in Figure 4.20

4.7.2   Tuning

Clearly, we do not want every processor to make a request for help before every serial recursive call, because this would result in the manager being flooded with messages towards the leaves of the recursion tree. As noted in Chapter 3, there is a minimum problem size below which it is not worth asking for help, because the time to send the arguments and receive the results is greater than the time to compute the result locally. This provides a simple lower bound, which can be compared to the result of the cost function described in Section 4.5: if the cost function for a particular recursive call (that is, the expected time taken by the call) returns a result less than the lower bound, the processor does not request help.

The lower bound therefore acts as a tuning function for the load-balancing system. Since it is dependent on the algorithm, architecture, MPI implementation, problem size, machine size, and input data, it can be supplied as either a compile-time or run-time parameter. I found reasonable approximations of this value for each divide-and-conquer function using a simple program that measures the time taken to send and receive the arguments and results for a function of size n between two processors, and that also measures the time taken to perform the function. The program then adjusts n up or down appropriately, until the two times are approximately equal. This provides a rough estimate of the problem size below which it is not worth asking for help. Chapter 6 further discusses the choice of this tuning value.

Note that most divide-and-conquer functions, which have monotonically decreasing subproblem sizes, act as self-throttling systems using this load-balancing approach. Initially, request traffic is low, because the subproblems being worked on are large, and hence there is a large time delay between any two requests from one processor. Similarly, there is little request traffic at the end of the algorithm, because all the subproblems have costs below the cutoff

71
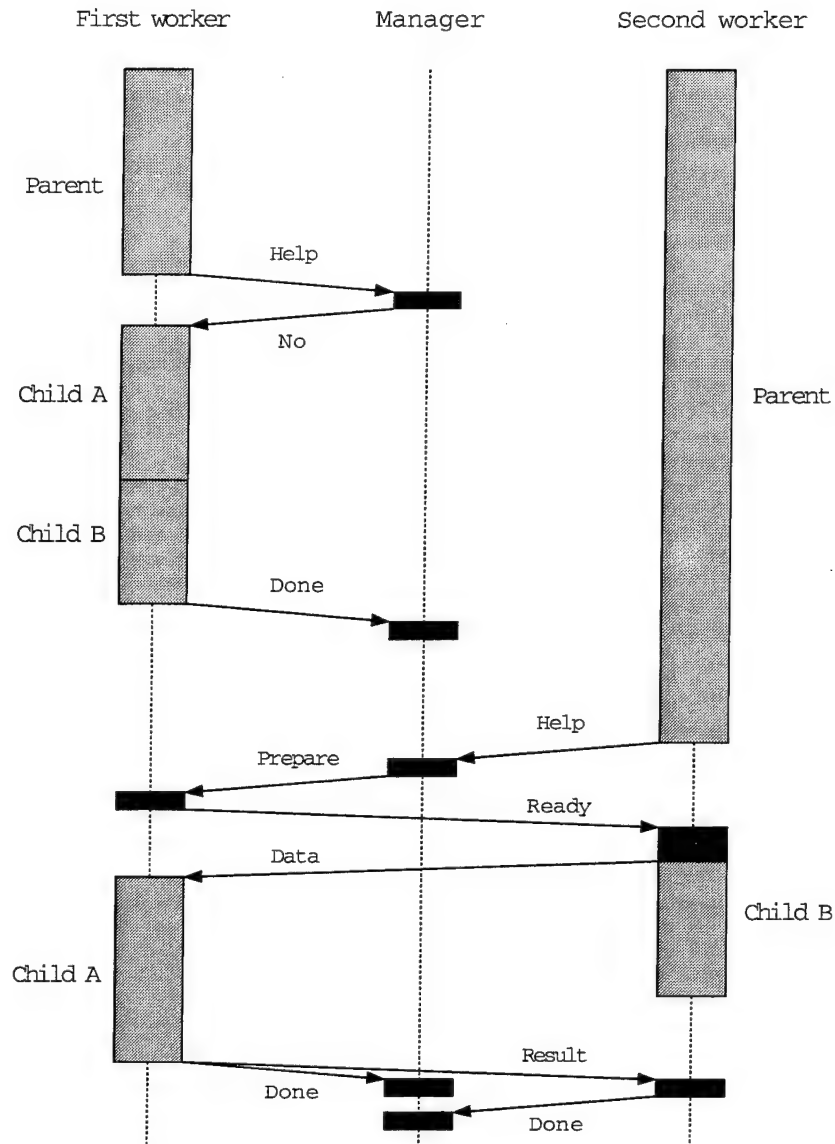
Figure 4.20: Load balancing between two worker processors and a manager. Time runs vertically downwards. Each worker processor executes a "parent" function call that in turn makes two "child" function calls. The first worker asks for help with its first child call from the manager but doesn't get it. After it has finished, the second worker asks for help. As the first worker is idle, it is instructed to help the second worker, which ships over its first child call and proceeds with its second.

limit. Requests are made when they can best be filled, towards the middle of an unbalanced algorithm, when some processors have finished but others are still working.

4.7.3 Data transfer

To be able to ship the arguments and results of function calls between processors, they must be converted into MPI messages. For every divide-and-conquer function, the preprocessor therefore generates two auxiliary functions. One, given the same arguments as the divide-and-conquer function, wraps them up into messages and sends them to a specified processor. The other has the same return type as the divide-and-conquer function; when called, it receives the result as a message from a specified processor.

Scalar arguments are copied into a single untyped buffer before sending. This enables the runtime system to send all of the scalar arguments to a function in a single message, incurring only the overhead of a single message. However, vector arguments are sent as individual messages. The reason for this is that we expect the vectors to be comparatively long, and their transmission time to therefore be dominated by bandwidth instead of latency. The act of copying separate vectors into a single buffer for sending would require more time than the latency of the extra messages.

The act of transmitting function arguments to a processor, and receiving results from it, is effectively a remote procedure call [Nel81]. Here the RPC is specialized by restricting it to a single divide-and-conquer function that can be called at any instant in the program.

4.8 Summary

In this chapter I have described the Machiavelli system, a particular implementation of the team parallel model for distributed-memory systems. Machiavelli is presented as an SPMD extension to C, using a block-distributed vector as its basic data structure, and relying on both predefined and user-defined data-parallel functions for its computation. I have described the implementation of Machiavelli in terms of a few key primitives, and outlined the protocol used to support demand-driven load-balancing. In addition I have explained the use of unbalanced vectors to avoid unnecessary communication.

Chapter 5

Performance Evaluation

> *Men as a whole judge more with their eyes than with their hands.*
> —Machiavelli

This is the first of two chapters that evaluate the performance of the Machiavelli system through the use of benchmarks. In this chapter I concentrate on benchmarking the basic primitives provided by Machiavelli; in the next chapter I demonstrate its performance on some algorithmic kernels.

There are three main reasons to benchmark Machiavelli's basic operations. The first is to demonstrate that they are suitable building blocks for a system that will support the claims of the thesis. To do this they must be efficiently scalable with both machine size and problem size, and they must be portable across different machine types. The second reason is to compare the performance of different primitives, both on the same machine and on different machines, in order to give a programmer information on their relative costs for use when designing algorithms. The final reason is to demonstrate how the performance of the primitives is related to their underlying implementation.

5.1 Environment

To prove the portability of Machiavelli, I chose two machines from extremes on the spectrum of memory coupling. The IBM SP2 is loosely coupled, being a distributed-memory architecture with workstation-class nodes connected by a switched message-passing network. I benchmarked on thin nodes, using `xlc -O3` and MPICH 1.0.12. In contrast, the SGI Power Challenge is a very tightly-coupled shared-memory architecture, with processors connected by a shared bus. I benchmarked on R8000 processors, using `cc -O2` and SGI MPI.

75

All floating-point arithmetic in these benchmarks uses double-precision 64-bit numbers, and all timings are the median of 21 runs on pseudo-random data generated with different seeds. A distributed random-number generator is used, so that for a given seed and length the same data is generated in a distributed vector, regardless of the number of processors. This is important for large problem sizes, where a single processor cannot be used to generate all the data.

As noted in Chapter 1, Machiavelli is intended to be used for large problem sizes, where $N \gg P$. The benchmarks reflect this, with the maximum problem size for each combination of machine and benchmark being chosen as the largest power of two that could be successfully run to completion.

Given randomized data, there are four variables involved for any particular benchmark: problem size, number of processors, machine architecture, and time taken. It is impractical to plot this data using a single graph, or on a set of three-dimensional graphs, due to the difficulty of comparing the relative positions of surfaces in space. I have therefore chosen to take three slices through the 3D space of algorithm size, machine size, and time for each algorithm, and show a separate set of slices for each machine architecture. The three slices are:

- The effect of the number of processors on the time taken for a fixed problem size. The problem size is fixed at the largest that can be solved on a single processor, and the number of processors is varied. This results in a graph of number of processors versus time. This can be thought of as a worst case: as the machine size increases, each processor works on less and less data, and hence parallel overheads tend to dominate the performance.

- The effect of the problem size on the time taken for a fixed number of processors. The number of processors is fixed, and the problem size is varied, resulting in a graph of problem size versus time. This graph shows the complexity of the algorithm when all other variables are fixed.

- The effect of the problem size on the time taken, as the problem size is scaled with the number of processors. The problem size is scaled linearly with the number of processors (that is, the amount of data per processor is fixed as the number of processors is varied), resulting in a joint graph of problem size and number of processors versus time. For algorithms with linear complexity, this graph shows the effect of parallel overheads in the code. It also demonstrates whether we can efficiently scale problem size with machine size to solve much larger problems than can be solved on one processor.

76

5.2 Benchmarks

I have chosen to benchmark three major sets of parallel Machiavelli primitives. The first set compare simple collective operations such as scan and reduce, which are implemented directly using the corresponding MPI functions. For a given machine, the running time of these MPI functions should depend only on the number of processors. The second set compare operations that involve all-to-all communication of data amongst the processors. These involve multiple MPI operations, whose running time will have a significant dependency on the amount of data moved. The final set shows the effect on all-to-all communication of an optimization using the unbalanced vectors described in Section 4.4.2. All of the primitives benchmarked have sequential implementations whose cost is linear in the size of the input.

5.2.1 Simple collective operations

As shown in Chapter 4, the simplest Machiavelli operations involving inter-processor communication consist of local computation (typically of cost $O(n)$ for a vector of length n) plus a single MPI operation. This category of operations includes scans and reductions. By comparing the performance of these operations to ones that involve purely local computation (such as distribute) we can compare the cost of the inter-processor communication to that of local computation. A second way to illustrate this is to compare performance on a single processor, when Machiavelli uses sequential code, to that on two processors, when it uses parallel code and MPI operations.

Figures 5.1 and 5.2 show the performance of the scan, reduce, and distribute primitives on the IBM SP2 and SGI Power Challenge. The functions being benchmarked are scan_sum_double, reduce_sum_double, and distribute_double. The distribute primitive creates a vector and then iterates over it, writing a scalar argument to each element, as shown in Figure 4.9 on page 60. This is the minimum operation required to create a vector (i.e., the shortest possible loop). The reduce primitive iterates over a source vector, applying the appropriate operator to create a local reduction, and then calls an MPI reduction function to create a global reduction, as shown in Figure 4.4 on page 53. The scan primitive can be seen as a combination of the distribute and reduce primitives: it reads from a source vector, creates and writes to a destination vector, calls an MPI scan function, and then iterates over the destination vector for a second time, as shown in Figure 4.5 on page 54.

The first thing to notice about Figures 5.1 and 5.2 is that the shapes of the graphs are very similar on the two machines, in spite of the differences in their machine architectures. This can be attributed to the simplicity of the primitives; they are almost purely memory bound, with very little inter-processor communication.

Figure 5.1: Three views of the performance of `scan`, `reduce`, and `distribute` on the IBM SP2. At the top left is the time taken for a fixed problem size as the number of processors is varied. At the top right is the time taken for a fixed number of processors as the problem size is varied. At the bottom is the time taken as the problem size is scaled linearly with the number of processors.
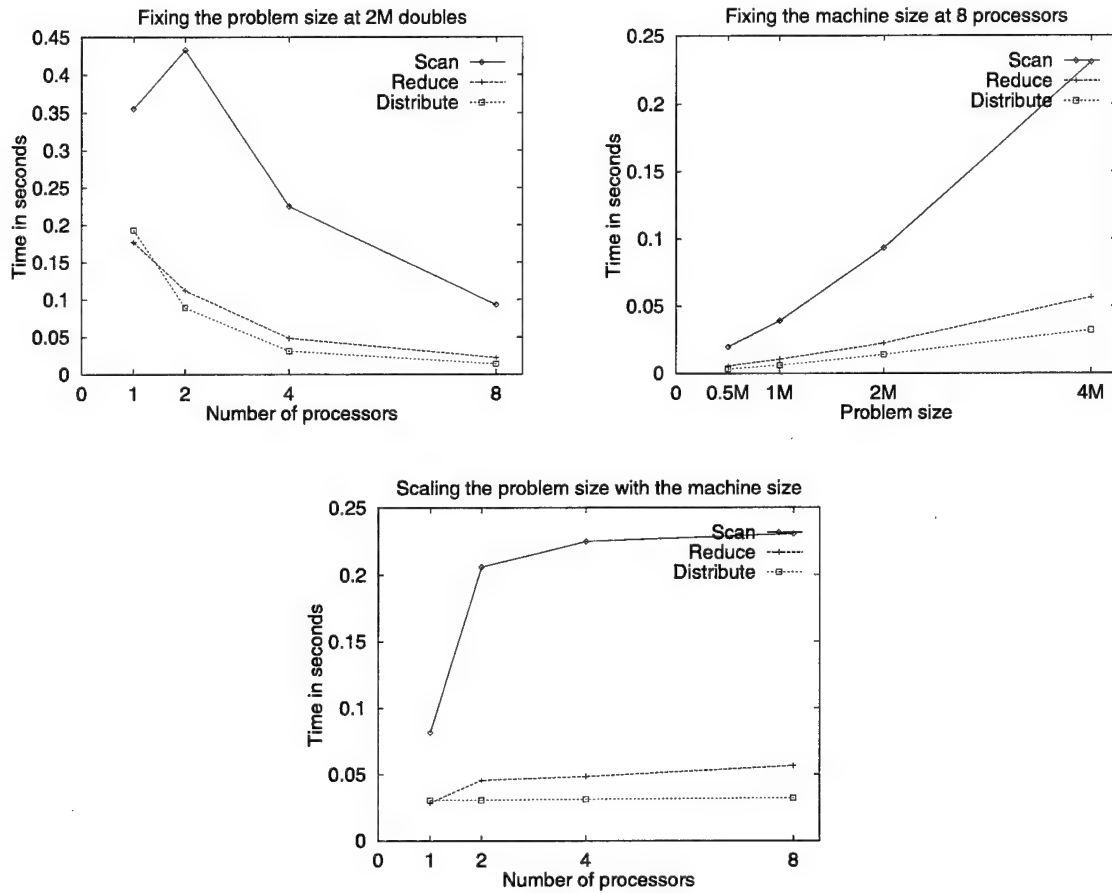
Figure 5.2: Three views of the performance of `scan`, `reduce`, and `distribute` on the SGI Power Challenge. Views are as in Figure 5.1. Note that fewer processors are available than on the SP2.

Looking at individual graphs, the performance of these primitives scales linearly with problem size for a given machine size, as we would expect based on their implementation. When the problem size is scaled with the machine size, we can see that the primitives are generally insensitive to the number of processors for large problem sizes. This indicates that the performance of the MPI reduce and scan operations called by the corresponding primitives are not a major bottleneck to scalability, since there is generally only a slight reduction in per-processor performance as the number of processors is increased. However, there is a major leap in the time taken by reduce and especially scan as we go from one to two processors. This corresponds to the switch between serial code (which doesn't need to call MPI primitives) and parallel code (which does). The distribute primitive does not need to call any MPI functions, and hence the performance of its serial and parallel versions is identical.

Finally, on both architectures the relationships between the costs of the primitives are the same. A scan is more costly than the sum of a distribute and a reduce, as we would expect based on their implementations (specifically, a parallel scan involves two reads and two writes to every element of a vector, compared to one read for a reduce and one write for a distribute). Also, the asymptotic performance of a unit-stride vector write (distribute) is slightly worse than that of a unit-stride vector read (reduce); this can be seen in the figures for single-processor performance. When more than one processor is used, the positions are reversed, with distribute becoming slightly faster than reduce because it does not involve any interprocessor communication.

5.2.2 All-to-all communication

The second set of benchmarks test all-to-all communication in the append and fetch primitives, which involve significant data transfer between processors. As explained in Section 4.3.3, these are two of the most complex Machiavelli primitives. The append primitive for two vectors calls four MPI all-to-all functions, two of size $O(p)$ to establish how many elements each processor will send and receive, and two of size $O(n)$ to transfer the data. The fetch primitive calls three MPI all-to-all functions; the first of size $O(p)$ establishes how many elements each processor will send and receive, the second of size $O(n)$ transfers the indices of the elements to fetch, and the third also of size $O(n)$ transfers the elements themselves. Benchmarking these two functions allows us to compare the cost of transferring contiguous data (in append) and scattered data (in fetch).

Figures 5.3 and 5.4 show three benchmark results for the IBM SP2 and SGI Power Challenge. The append benchmark appends two vectors of length $n/2$ to form a single vector of length n. The two fetch benchmarks fetch every element from a vector of length n according to an index vector to create a rearranged vector of length n. The first fetch benchmark uses an index vector containing $[0, 1, 2, \ldots, n-1]$, resulting in a null fetch. That is, it creates a result

80

Append and fetch on the IBM SP2
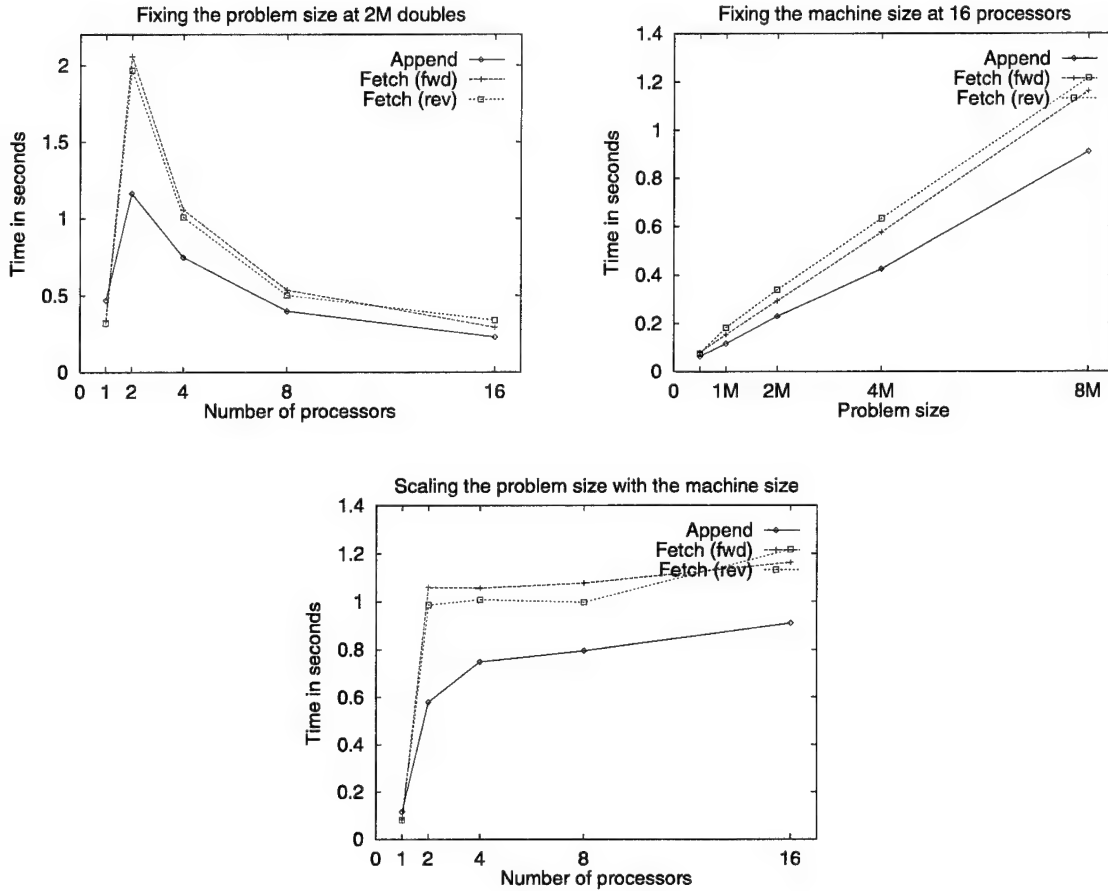
Figure 5.3: Three views of the performance of `append` and `fetch` on the IBM SP2. The `append` benchmark involves appending two vectors of length n. The `fetch` benchmarks involve fetching via an index vector (that is, a null operation), and fetching via a reverse index vector (that is, a vector reversal). Views are as in Figure 5.1

Append and fetch on the SGI Power Challenge
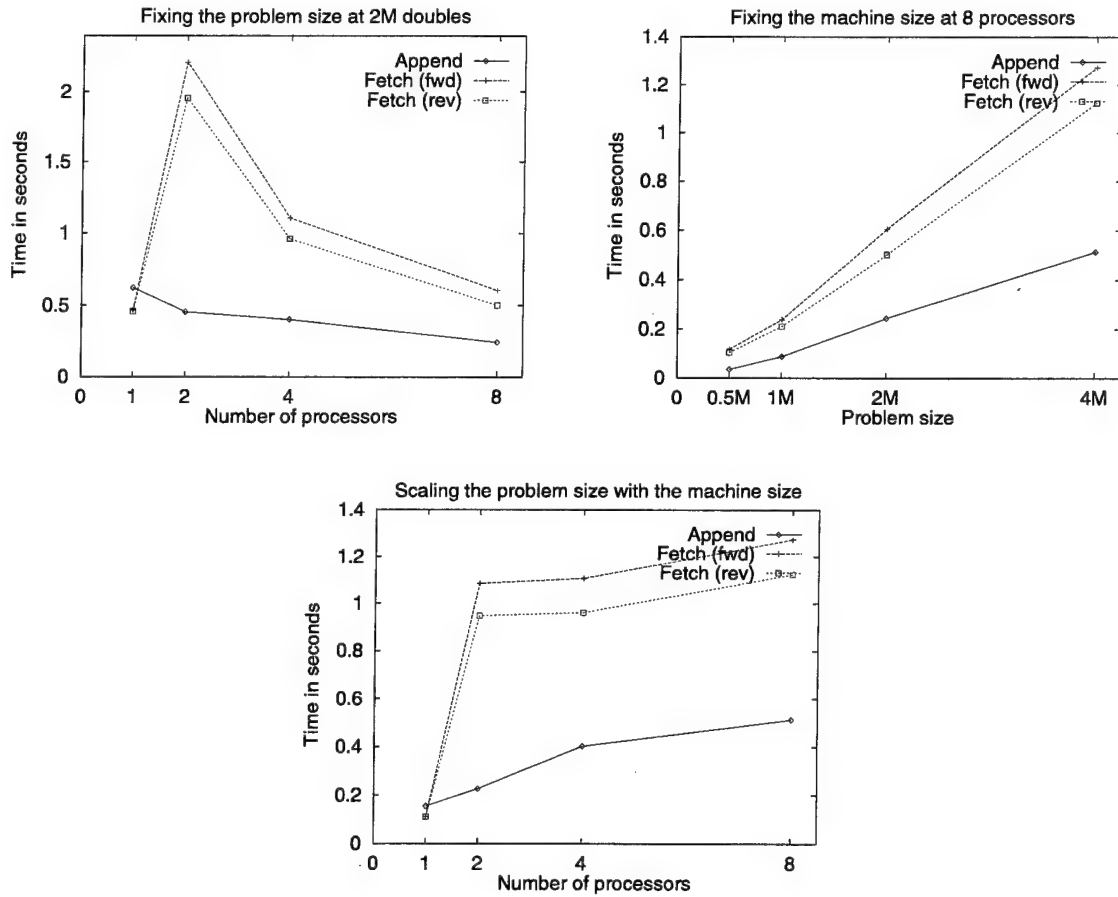

Figure 5.4: Three views of the performance of `append` and `fetch` on the SGI Power Challenge. Views are as in Figure 5.1.

vector containing the same elements in the same order as the source vector. This is a simple best-case input set; theoretically, no data needs to be transferred between processors. The second `fetch` benchmark uses an index vector containing $[n-1, n-2, n-3, \ldots, 0]$, resulting in a vector reversal operation. This could be more efficiently expressed as a `send`, but was chosen to represent a simple worst-case input set; for the even numbers of processors tested, every processor needs to send all its data and receive new data.

Again, the shape and relationships of the graphs are very similar on the two machines, with the exception of the `append` primitive, which is comparatively cheaper on the Power Challenge, and does not suffer from such a serious performance degradation on going from one to two processors as on the SP2. It is also interesting to note that `append` is faster than `fetch` on both machines. The reason for this is that in `append` we are dealing with contiguous chunks of data. Therefore, direct data pointers to the source and destination vectors can be passed to the MPI all-to-all functions. By contrast, `fetch` is not optimized for the case of contiguous data, and requires extra loops to copy indices and data items into and out of temporary buffers before calling MPI functions. The cost of this extra memory-to-memory traffic is higher than that of sending the data between processors. The significantly lower relative cost of append on the SGI is due to the implementation of all-to-all functions on the shared-memory architecture; messages can be "sent" between processors by transferring a pointer to the block of shared memory containing the message from the sending processor to the receiving processor. This makes data transfer between processors relatively cheap.

Compared to the `scan`, `reduce` and `distribute` primitives, `append` and `fetch` are typically 5–10 times more expensive, reflecting the cost of inter-processor data transfer. Additionally, they suffer from a greater parallel overhead, due to their more complex implementation and the exchange of messages in multiple all-to-all communication steps. This is reflected in their performance on fixed problem sizes; for example, on a minimum size problem the `fetch` primitive takes about as long on sixteen SP2 processors as it does on one. However, if the problem size is increased linearly with the machine size, scalability is much better.

Finally, there is little difference between the performance of the forward and reverse `fetch` benchmarks on the two machines, despite the fact that one has to move all of the data and the other theoretically has to move none. Again, this is due to the fact that the implementation of `fetch` has not been optimized for special cases, and treats fetching from the calling processor exactly the same as fetching from a separate processor, with loops to copy data into and out of buffers. It relies on the MPI implementation to short-circuit the case of a processor sending a message to itself. Based on inspection of the source code, MPICH does not do so until very late in the call chain, and the results for the SP2 therefore reflect only the difference between moving the data between processors and moving the data between buffers in memory. The proprietary SGI MPI implementation appears to do better in this respect, resulting in a larger performance gap. However, even on this machine the performance of `fetch` primitive is still relatively insensitive to the exact pattern of the index vector.

5.2.3 Data-dependent operations

The third set of benchmarks are intended to show both the variation in performance of a data-dependent Machiavelli primitive, namely `even`, and the effects of unbalanced vector optimizations. As described in Section 4.3.4, `even` is an "extract even elements" primitive that has been generalized to allow for arbitrary element sizes (for example, rows of a matrix). It is implemented with a simple copying loop that discards odd elements of the source vector, resulting in an unbalanced vector in parallel code. Subsequent Machiavelli primitives may operate on the vector in its unbalanced state, or may first use a `pack` primitive to balance it across the processors, as explained in see Section 4.4.2.

Figures 5.5 and 5.6 show three benchmark results for `even` on the IBM SP2 and SGI Power Challenge. The first benchmark uses a block size of one. With the vector lengths and machine sizes chosen, this requires no interprocessor communication. The second benchmark uses a block size of $n/2$ for a vector of length n; that is, it returns the first half of the vector, and will do so in an unbalanced form when run on more than one processor. The third benchmark also uses a block size of $n/2$, but adds a `pack` operation to balance the resulting vector. This assumes that subsequent operations need to operate on the vector in a balanced state.

Once again, the shape of the graphs are very similar on the two architectures. The time taken by the primitives scales linearly with problem size for a fixed number of processors on the SP2, and shows relatively good scalability as the number of processors is increased. However, the SGI fares less well at large problem sizes, possibly due to contention on the shared memory bus.

Note that it is faster on one processor to execute `even` with a block size of $n/2$ than with a block size of 1. Although both read and write $n/2$ elements, the former uses a stride of 1, while the latter uses a stride of 2, which is slower on current memory architectures. On two processors (without packing the resulting vector) it is faster to execute `even` with a block size of 1, since both processors read and write $n/4$ elements; when using a block size of $n/2$, one processor reads and writes all of the elements.

As we would expect, the inter-processor communication of a `pack` operation considerably increases the cost of `even` if it is necessary to balance the result, resulting in a slowdown of 2–5 times. This shows the performance advantages of leaving newly-generated vectors in an unbalanced state where possible.

5.2.4 Vector/scalar operations

The final benchmark emphasizes the greatest possible performance difference between serial and parallel code, through the use of the Machiavelli `get` primitive. This returns the value of a specified element in a vector to all processors, as described in Section 4.3.4. In serial code

84

Even on the IBM SP2

Figure 5.5: Three views of the performance of the `even` primitive, which returns all elements at an "even" prefix on the IBM SP2. The three benchmarks are with a stride of 1, with a stride of $n/2$ and leaving the result as an unbalanced vector, and with a stride of $n/2$ plus packing the result to remove the imbalance. Views are as in Figure 5.1.
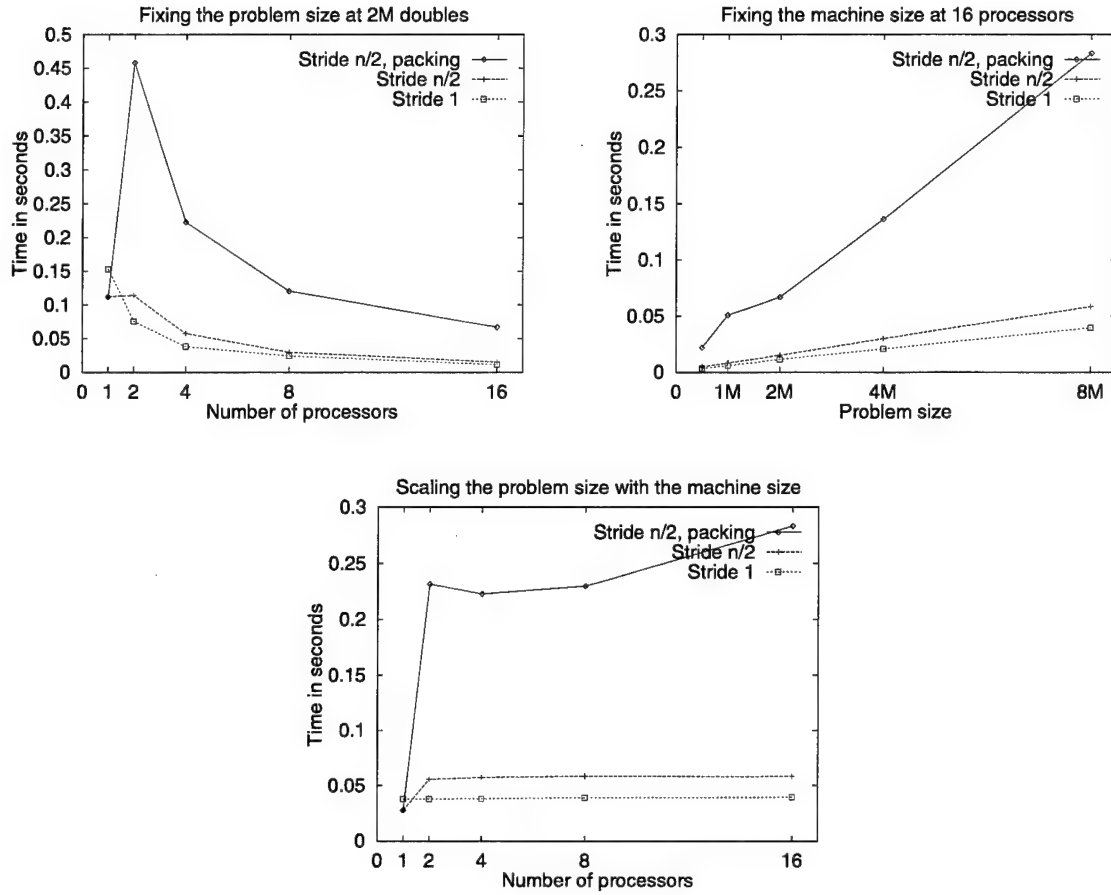
85

Even on the SGI Power Challenge
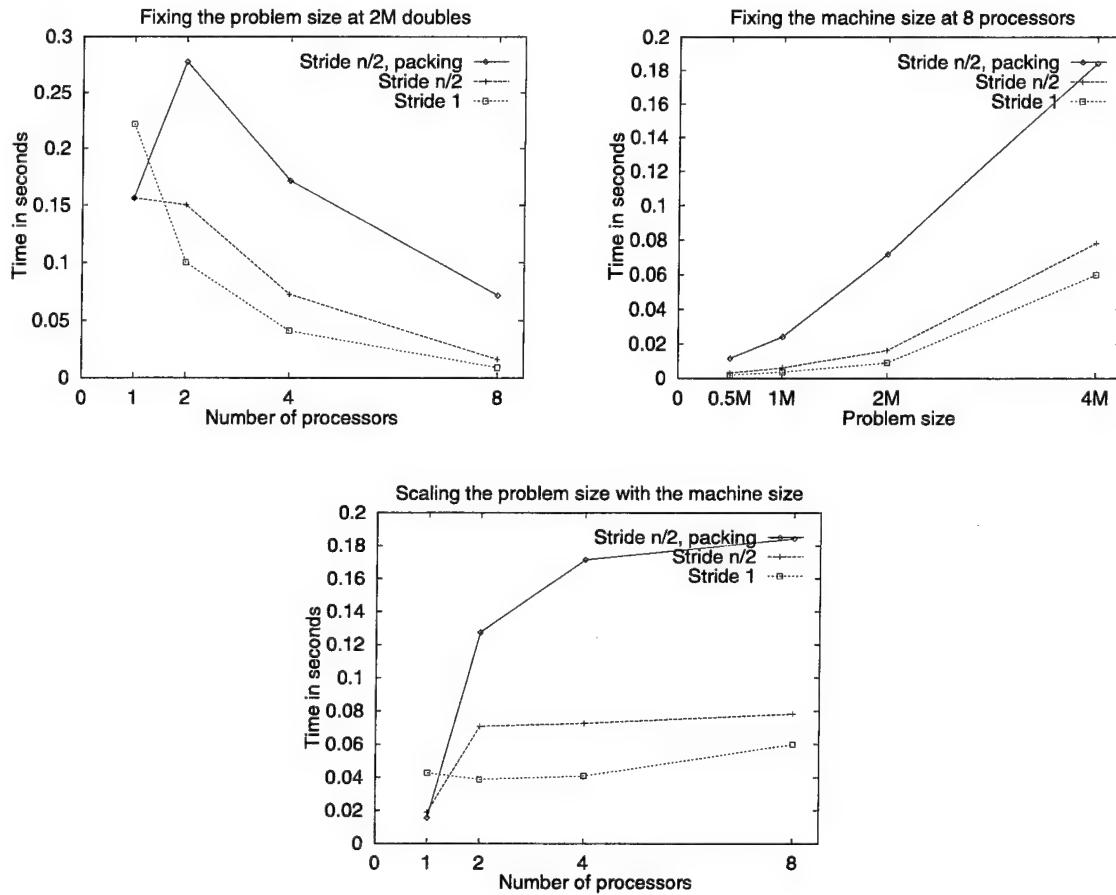
Figure 5.6: Three views of the performance of the `even` primitive, which returns all elements at an "even" prefix on the SGI Power Challenge. Views are as in Figure 5.1.
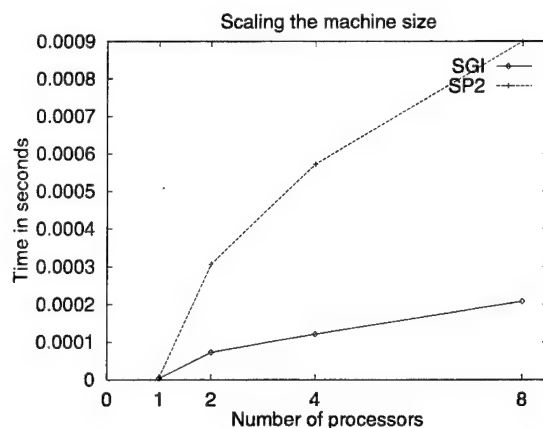
Figure 5.7: Performance of the `get` primitive on the IBM SP2 and SGI Power Challenge.

this reduces to a single memory read. However, in parallel code it requires a broadcast from the processor that contains the specified element. This is implemented using an MPI broadcast operation, as shown in Figure 4.6 on page 58. As such, its performance is dependent only on the number of processors, and not on the data size.

Figure 5.7 shows the performance of the `get` primitive on both architectures. Note that while the Power Challenge is 4–5 times faster than the SP-2 when running parallel code, and that this is a much cheaper primitive than the ones seen previously, both machines are more than two orders of magnitude slower when running parallel code than the single memory read in serial code. This is an extreme example to show that data access in Machiavelli, as in any data-parallel language, should generally be performed using the data-parallel primitives, rather than trying to emulate them using a loop wrapped around a call to `get`, for example.

5.3 Summary

In this chapter I have benchmarked a variety of Machiavelli primitives on two parallel machines with very dissimilar architectures. Despite this, their performance on the various primitives was generally comparable.

For all primitives whose parallel versions involve inter-processor communication, per-processor performance falls as we move from serial code running on one processor to parallel code running on multiple processors, reflecting the increased cost of accessing remote memory versus local memory. For primitives involving all-to-all communication, this fall can be significant, and severely limits the ability of Machiavelli to achieve useful parallelism on small problem sizes. However, all primitives show better per-processor performance as the problem

87

size is scaled with the machine size. In general, the cost per element of an operation involving all-to-all communication is 5–10 times that of an operation where no communication takes place.

I have also used these benchmarks to demonstrate an extreme case of the performance gains possible when using the unbalanced vector optimization described in Section 4.4.2, and the greatest performance difference possible between parallel and serial versions of the same code, which occurs in the `get` primitive.

Chapter 6

Expressing Basic Algorithms

Whenever you see the word "optimal" there's got to be something wrong.
—Richard Karp, WOPA'93

This chapter provides further experimental results to support my thesis that irregular divide-and-conquer algorithms can be efficiently mapped onto distributed-memory machines. Specifically, I present results for four algorithmic kernels:

Quicksort: Sorts floating-point numbers. This is the classic quicksort algorithm, but expressed in a high-level data-parallel form rather than the standard low-level view of moving pointers around in an array.

Convex hull: Finds the convex hull of a set of points on the plane. This is an important substep of more complex geometric algorithms, such as Delaunay triangulation (see Chapter 7).

Geometric separator: Decomposes a graph into equally-sized subgraphs such that the number of edges between subgraphs is minimized. This is an example of an algorithm that requires generalized all-to-all communication.

Matrix multiply: Multiplies two dense matrices of floating-point numbers. This is an example of a numerically intensive algorithm with a balanced divide-and-conquer style.

I aim to show three things in this chapter. First, that divide-and-conquer algorithms can be expressed in a concise and natural form in the team model in Machiavelli. I give both the NESL code and the Machiavelli code for each algorithm. After allowing for the extra operations required by Machiavelli's structure as a syntactic preprocessor on top of the C language—specifically, variable declarations, memory deallocation, and operation nesting limits—the NESL and Machiavelli code typically have a one-to-one correspondence.

89

Second, to describe the effect of various optimizations. Those hidden by Machiavelli include unbalanced vectors and active load balancing. Those exposed to the user include supplying more efficient serial code where a faster algorithm is available (in the case of quicksort and matrix multiplication), restructuring an algorithm to eliminate vector replication (in the case of the convex hull algorithm), and using both serial and parallel versions of compiled code to generate faster algorithms (in the case of the median substep of the geometric separator).

My final goal is to demonstrate good absolute performance compared to an efficient serial version of each algorithm. Note that for most of the algorithms given, the total space required by the parallel algorithm is no more than a constant factor greater than that required by the serial algorithm (if a serial machine large enough to solve the problem were available).

Variants of each algorithm are chosen to show the effects of load balancing, unbalanced vectors, and specialized serial code. Note that load-balancing can be selected at compile time, and unbalanced vectors are automatically provided by Machiavelli, but that specialized serial code must be supplied by the user.

In addition to the IBM SP2 and SGI PowerChallenge described in Chapter 5, the algorithms were also run on a distributed-memory Cray T3D, using `cc -O2` and MPICH 1.0.13. The same set of three graphs is shown for each combination of algorithm and machine.

6.1 Quicksort

The NESL and Machiavelli code for quicksort are shown in Figure 4.1 on page 48; note that their structure and style are similar. Quicksort is an unbalanced divide-and-conquer algorithm with a branching factor of two, a data-dependent divide function (selecting the pivot), but a size function that is independent of the data (the sum of the sizes of the three subvectors is equal to the size of the input vector). Data parallelism is available in the apply-to-each operations and in the append function. The quicksort algorithm has expected complexities of $O(n \log n)$ work and $O(\log n)$ depth, although a pathological sequence of pivots can require $O(n^2)$ work and $O(n)$ depth.

Quicksort is an extreme case for the team-parallel model in two respects. First, it performs very little computation relative to the amount of data movement. Therefore, the overhead of time spent distributing vectors between processors is proportionately higher than in other algorithms. This effect is most pronounced when a small problem is solved on many processors. Second, a very simple and efficient serial algorithm is available. This uses in-place pointer operations, and is significantly faster than vector code compiled for a single processor, which must create and destroy vectors in memory.

The major computation of the algorithm lies in creating the three vectors `les`, `eql`, and `grt`. The Machiavelli preprocessor fuses the three loops that create these vectors, resulting in

optimal vector code. In the parallel code, the resulting vectors are unbalanced. The `split` function then uses a single call to `MPI_Alltoallv()` to redistribute this data, as described on page 66. Finally, the `append()` function can again use unbalanced vectors, this time to replace $\log P$ intermediate append steps on the way back up the recursion tree with a single append at the top level that combines data from all P processors at once.

Figures 6.1 and 6.2 show the performance of quicksort on the Cray T3D and IBM SP2, respectively. The four variants of the algorithm are: without load balancing, with load balancing, with efficient serial code supplied by the user (specifically, that shown in Figure 4.17 on page 68), and with efficient serial code plus load balancing. Once again, the shape and relationship of the graphs on the two architectures is very similar. We can immediately see the effect of the user-supplied serial code, which is approximately 5–7 times faster than the vector code on one processor, depending on the machine architecture. However, since it is still necessary to use parallel code on more than one processor, the performance of the variants with efficient serial code drops off more sharply as the number of processors is increased. By comparison, the performance of the vector code is much more consistent as the number of processors is increased, especially when load balancing is used. Thus for large numbers of processors, the performance advantage of the user-supplied serial code over the vector code falls to a factor of 2–3 times faster.

Load balancing also has a greater effect on the vector code than on the user-supplied serial code, increasing the performance of the former by up to a factor of two, but the latter by only a factor of 1.1–1.3. This reflects the greater relative cost of load balancing when efficient user-supplied code is used (see also Section 6.1.1). The cross-over point for load balancing on quicksort, at which the overall improved performance outweighs the loss of one processor as a dedicated manager, takes place between 4 and 8 processors for all architectures and algorithm variants. Note that a better choice of pivot values—for example by taking the median of a constant number of values—would improve the performance of the unbalanced code and would therefore reduce the effect of load balancing.

All four variants of the algorithm scale very well with problem size on a fixed number of processors, and all but the unbalanced vector code perform well as the machine size is scaled with the problem size. Indeed, the performance per processor of the balanced vector code actually increases as both the problem size and number of processors are increased, in spite of the fact that this is an $O(n \log n)$ algorithm. This is due to the increasing effectiveness of load balancing with machine size; with more processors in use, the likelihood that one will be available to assist another in load balancing also increases.

Overall, by using user-supplied serial code we can maintain a parallel efficiency of 22-30%, even for small problem sizes on large numbers of processors. Since as previously mentioned quicksort is an extreme case, this can be considered as a likely lower bound on the performance of Machiavelli on divide-and-conquer algorithms.

Quicksort on the Cray T3D
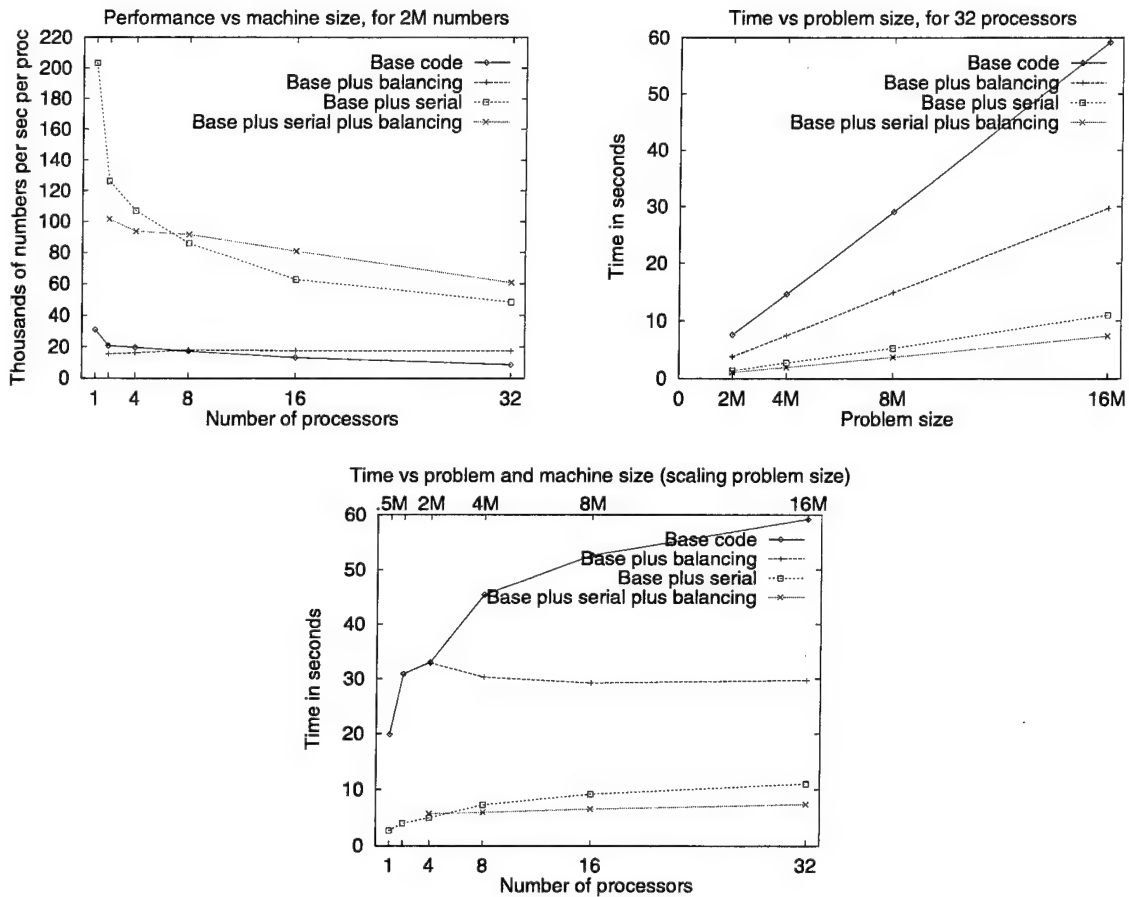
Figure 6.1: Three views of the performance of quicksort in Machiavelli on the Cray T3D. At the top left is the per-processor performance for a fixed problem size, as the number of processors is varied. At the top right is the time taken for a fixed number of processors, as the problem size is varied. At the bottom is the time taken as the problem size is scaled with the number of processors (from 0.5M on 1 processor to 16M on 32 processors). The four variants of the algorithm are: the basic vector algorithm, the basic algorithm plus load balancing, the basic algorithm with user-supplied serial code, and the basic algorithm with user-supplied serial code plus load balancing.

Quicksort on the IBM SP2
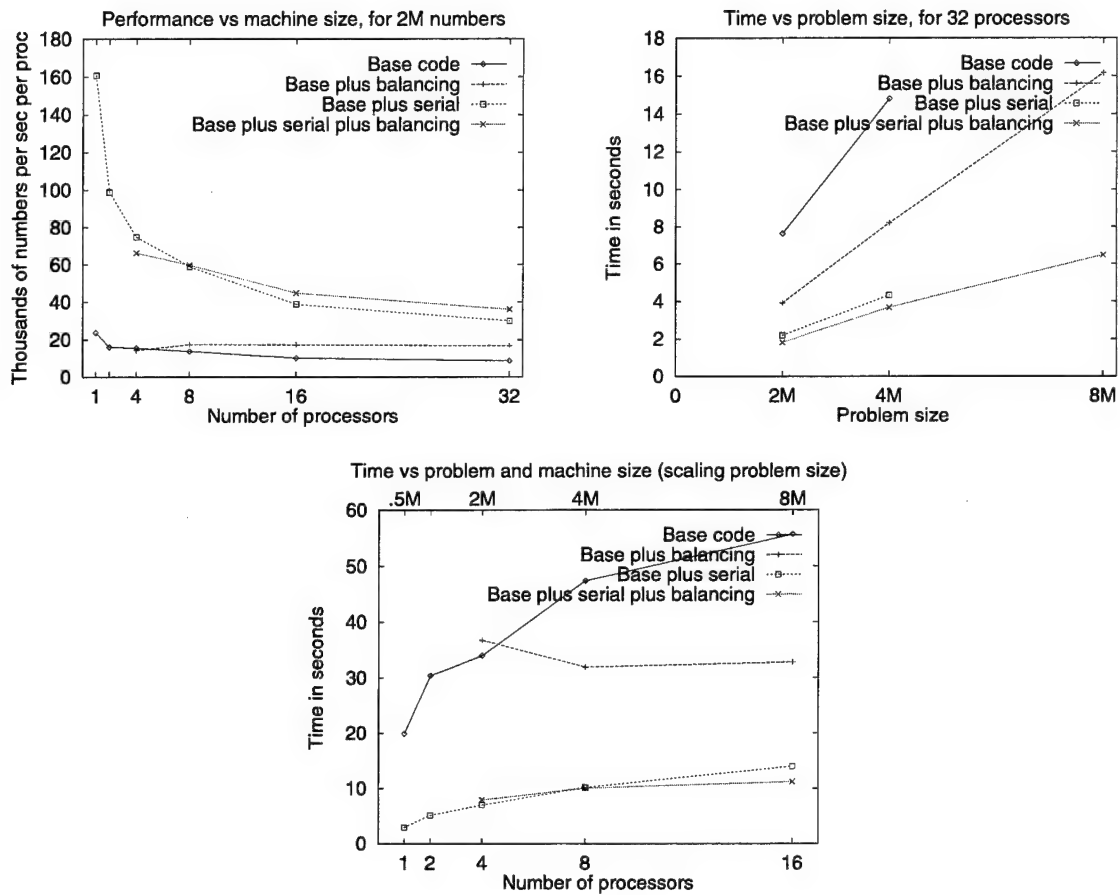
Figure 6.2: Three views of the performance of quicksort in Machiavelli on the IBM SP2. The views are as in Figure 6.1.
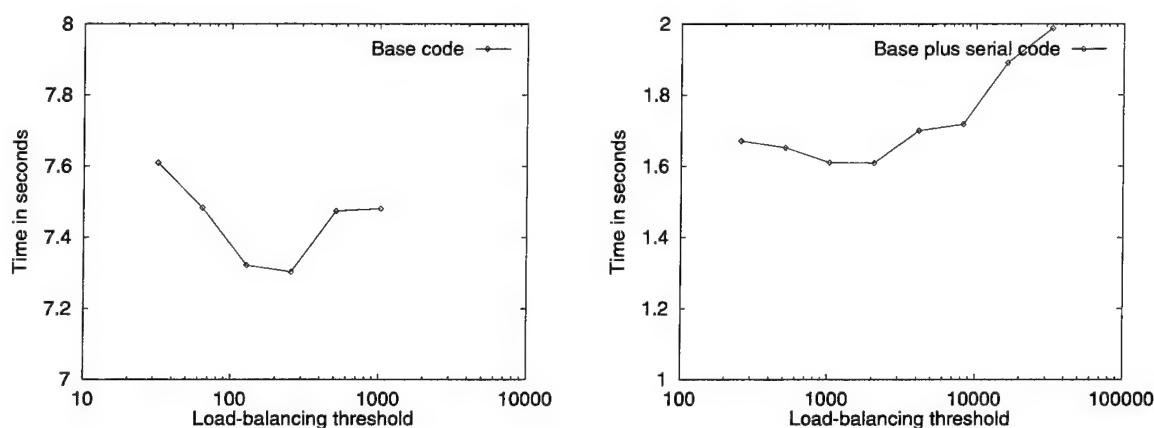
Figure 6.3: The time taken by Machiavelli quicksort on the Cray T3D, against the load-balancing threshold, for serial code generated by the preprocessor (left) and supplied by the user (right). 2M numbers are being sorted on 16 processors. Note that there is an optimal threshold: below it, processors spend too much time requesting help, while above it, processors do not request help soon enough.

6.1.1 Choice of load-balancing threshold

The choice of the threshold problem size above which a processor running serial code requests help from the load-balancing system, has a small but noticable effect on overall performance. Figures 6.3 and 6.4 show the effect of this load-balancing threshold on the performance of quicksort on the Cray T3D and IBM SP2, for both vector code and for the user-supplied serial code. Two effects are at work here. As the threshold approaches zero, the client processors spend all their time requesting help from the processor acting as manager, which involves waiting for a minimum of two message transmission times. On the other hand, as the threshold approaches infinity, the client processors never request help, and the overall behavior is exactly the same as that of the unbalanced algorithm, but with one less processor doing useful work. There is therefore an optimal threshold between these two extremes.

Note that the faster user-supplied serial code has a higher optimal threshold, since it can perform more useful work in the time it would otherwise take to request help from the manager. Also, both architectures exhibit a "double-dip" in the graph, although it is much more pronounced on the SP2. This corresponds to the point at which the MPICH implementation switches message delivery methods, changing from an "eager" approach for short messages to a "rendezvous" approach for longer messages. This affects the time taken to perform load-balancing between processors, and hence the optimal threshold.

94
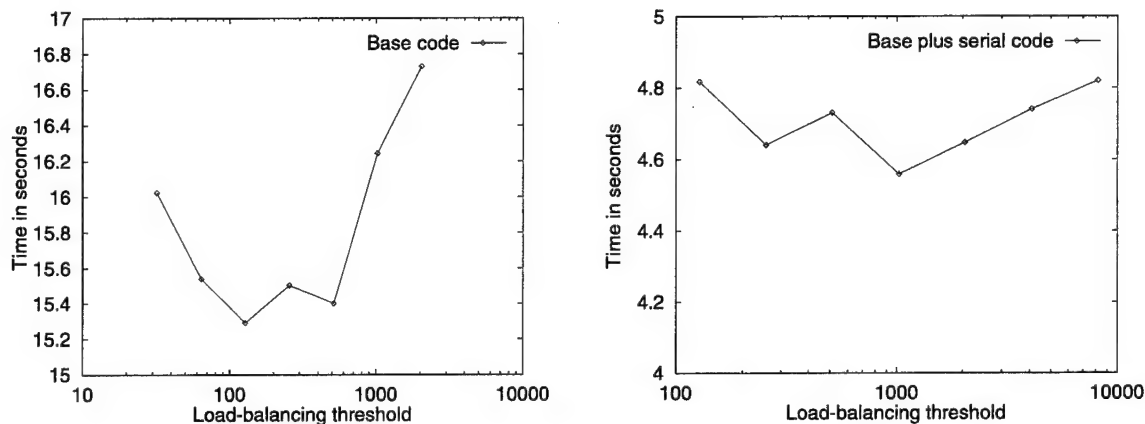
Figure 6.4: The time taken by Machiavelli quicksort on the IBM SP2, against the load-balancing threshold, for serial code generated by the preprocessor (left) and supplied by the user (right). 2M numbers are being sorted on 16 processors. Note the "double-dip" effect, due to a change in the way the MPI implementation delivers messages as the message size increases.

6.2 Convex Hull

The *quickhull* algorithm [PS85], so named for its similarity to quicksort, finds the convex hull of a set of points on the plane. Figures 6.5 and 6.6 show the algorithm expressed in NESL and Machiavelli respectively. Again, there is a direct and obvious line-by-line relationship between the NESL code and the Machiavelli code, although the NESL function to compute cross products is inlined into the Machiavelli code using a macro.

The core of the algorithm is the `hsplit` function, which recursively finds all the points on the convex hull between two given points on the hull. The function works by discarding all the points to one side of the line formed by the two points, finding the point furthest from the other side of the line (which must itself be on the convex hull), and using this point as a new endpoint in two recursive calls. This function therefore has a branching factor of two, and data dependencies in both the divide function (finding the furthest point), and the size function (discarding points based on the furthest point). The entire convex hull can be computed using two calls to `hsplit`, starting from the points with the minimum and maximum values of one axis (which must be on the convex hull), and ending at the maximum and minimum, respectively. Overall, the algorithm has expected complexities of $O(n \log n)$ work and $O(\log n)$ depth. As with quicksort, a pathological data distribution can require $O(n^2)$ work and $O(n)$ depth. However, in practise the quicksort algorithm is typically faster than more complex convex hull algorithms that are guaranteed to perform only linear work, because it is such a simple algorithm with low constant factors [BMT96].

95

```
% Returns the distance of point o from line. %
function cross_product(o,line) =
let (xo,yo) = o;
    ((x1,y1),(x2,y2)) = line;
in (x1-xo)*(y2-yo) - (y1-yo)*(x2-xo);

% Given points p1 and p2 on the convex hull, returns all the
  points on the hull between p1 and p2 (clockwise), inclusive
  of p1 but not of p2. %
function hsplit(points,(p1,p2)) =
let cross = {cross_product(p,(p1,p2)): p in points};
    packed = {p in points; c in cross | plusp(c)};
in if (#packed < 2) then [p1] ++ packed
   else
      let pm = points[max_index(cross)];
      in flatten({hsplit(packed,ends) : ends in [(p1,pm),(pm,p2)]});

function quick_hull(points) =
let x = {x : (x,y) in points};
    minx = points[min_index(x)];
    maxx = points[max_index(x)];
in hsplit(points,minx,maxx) ++ hsplit(points,maxx,minx);
```

Figure 6.5: NESL code for the quickhull convex hull algorithm, taken from [Ble95].

The C code generated by the Machiavelli preprocessor is similar to that generated from quicksort, with the loops generating the vectors `cross` and `packed` being fused, and the `packed` vector being left in an unbalanced state. Note that the `quick_hull()` function, while not in itself recursive, nevertheless contains calls to `hsplit()` using the `split` syntax, which results in two teams being formed to execute the two invocations of `hsplit()`.

6.2.1 Eliminating vector replication

Note that in `hsplit()` the `split()` function must now send the vector `packed` to both recursive calls. In parallel code this results in replication of the vector, with a copy being sent to the two subteams of processors. This requires a second call to `MPI_Alltoallv()`, and increases the memory requirements of the parallel code. Additionally, in the serial code it results in two loops being performed over each instance of `packed`, instead of the optimal one.

These problems can be solved by modifying the code as shown in Figure 6.7. The computation has been "hoisted" through the recursion, so that we precompute the vectors in advance. This doubles the amount of computation code. However, the four apply-to-each operations are all fused (thereby improving the performance of the algorithm in both serial and parallel code), and arguments to functions are no longer replicated, reducing memory requirements and parallel communication bandwidth.

The effect of this can be seen in Figures 6.8, 6.9 and 6.10, which show the performance of the quickhull algorithm on the Cray T3D, IBM SP2, and SGI Power Challenge, respectively.

96

```
#define CROSS_PRODUCT(P, START, FINISH) \
        (((START.x - P.x) * (FINISH.y - P.y)) - \
         ((START.y - P.y) * (FINISH.x - P.x)))


vec_point hsplit (vec_point points, point p1, point p2)
{
  vec_point packed, result, left, right;
  vec_double cross;
  point pm;

  cross = { CROSS_PRODUCT (p, p1, p2) : p in points };
  packed = { p : p in points, c in cross | c > 0.0 };

  if (length (packed) < 2) {
    result = append (vector (p1), packed);
    free (cross); free (packed);
  } else {
    pm = get (points, reduce_max_index (cross));
    free (cross);
    split (left = hsplit (packed, p1, pm),
           right = hsplit (packed, pm, p2));
    result = append (left, right);
    free (packed); free (left); free (right);
  }
  return result;
}


vec_point quick_hull (vec_point points)
{
  vec_point left, right, result;
  vec_double x_values;
  point minx, maxx;

  x_values = { p.x : p in points };
  minx = get (points, reduce_min_index (x_values));
  maxx = get (points, reduce_max_index (x_values));
  split (left = hsplit (points, minx, maxx),
         right = hsplit (points, maxx, minx));
  result = append (left, right);
  free (left); free (right);
  return result;
}
```

Figure 6.6: Machiavelli code for the quickhull convex hull algorithm. Note that the vector `packed` is passed to both recursive calls in `hsplit`, wasting space in serial code and causing replication in parallel code. Compare to the NESL code in Figure 6.5.

97

```
vec_point hsplit_fast (vec_point points, point p1, point p2, point pm)
{
  vec_point packedl, packedr, result, left, right;
  vec_double crossl, crossr;
  int max_indexl, max_indexr;
  point pml, pmr;

  if (length (points) < 2) {
    result = append (vec (p1), points);
    free_vec (points);
  } else {
    crossl = { CROSS_PRODUCT (p, p1, pm) : p in points };
    crossr = { CROSS_PRODUCT (p, pm, p2) : p in points };
    packedl = { p : p in points, c in crossl | c > 0.0 };
    packedr = { p : p in points, c in crossr | c > 0.0 };
    pml = get (points, reduce_max_index (crossl));
    pmr = get (points, reduce_max_index (crossr));
    free (crossl); free (crossr); free (points);
    split (left = hsplit_fast (packedl, p1, pm, pml),
           right = hsplit_fast (packedr, pm, p2, pmr));
    result = append (left, right);
    free (left); free (right);
  }
  return result;
}
```

Figure 6.7: A faster and more space-efficient version of `hsplit`. We precompute `packed` for the two recursive calls. This saves both time and space.

The input set consists of points whose x and y coordinates are chosen independently from the normal distribution of points. The three variants of the algorithm are: with the initial version of `hsplit()`, with the rewritten "inverted" version of `hsplit()`, and with the inverted version of `hsplit()` plus a "lazy" append optimization.

6.2.2 Eliminating unnecessary append steps

The lazy append optimization exploits the fact that the only action of the algorithm on returning up the recursion tree is to append the two resultant vectors together. Since the vectors originate on individual processors, the action of $O(\log p)$ append operations to incrementally concatenate p vectors into one vector spread across p processors can be replaced by one append operation at the top level, which combines the p vectors in a single operation.

As we would expect, the "lazy" append has an increasing impact on performance as the number of processors is increased, since it eliminates more and more append steps. However, the problem size has very little effect on the performance improvement of lazy append, since the size of the final convex hull is small compared to the large input sets, and is relatively constant for a normal distribution. This explains the near-constant performance improvement of lazy append on the T3D and SP2 when the machine size is fixed and the problem size

98

Convex hull on the Cray T3D
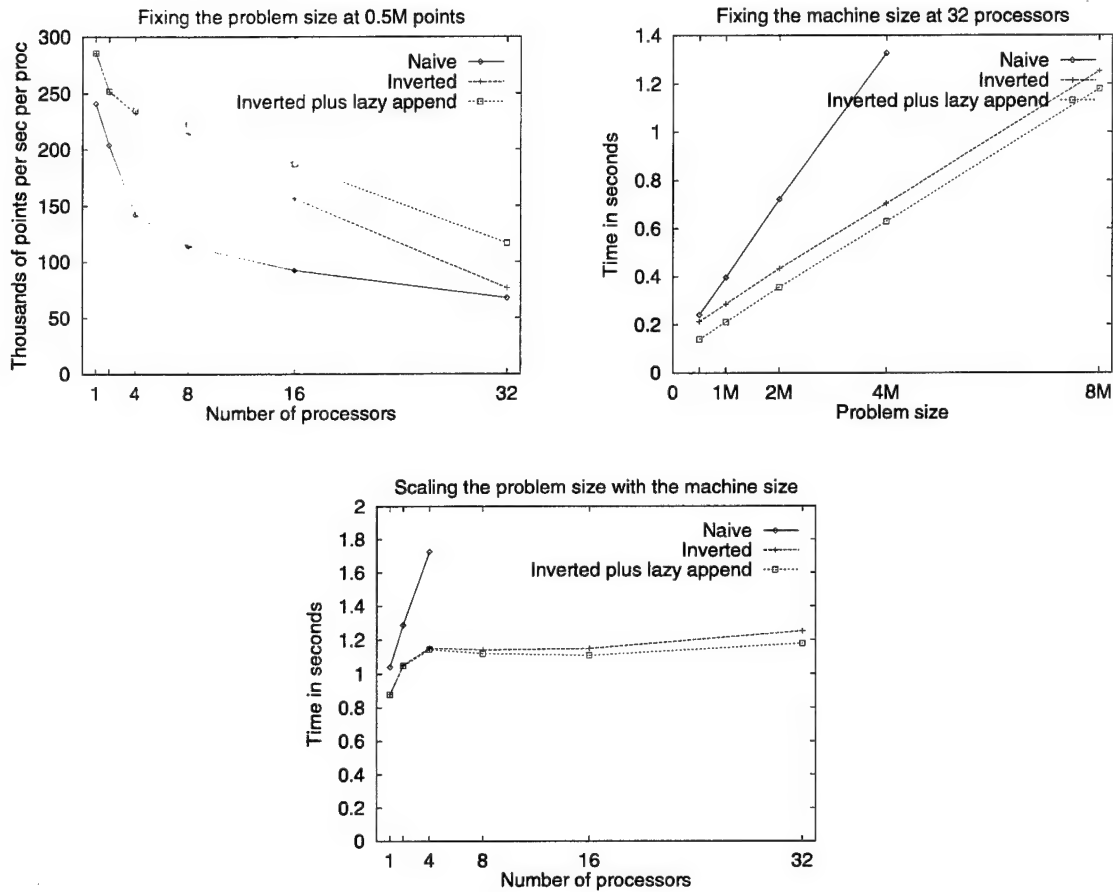
Figure 6.8: Three views of the performance of convex hull in Machiavelli on the Cray T3D. At the top left is the per-processor performance for a fixed problem size (0.5M points), as the problem size is varied. At the top right is the time taken for 32 processors, as the problem size is varied. At the bottom is the time taken as the problem size is scaled with the number of processors (from 0.25M points on one processor to 8M on 32 processors). The three variants of the algorithm are: the basic algorithm, the "inverted" algorithm with a more efficient `hsplit()` function, and the inverted algorithm plus a "lazy" append operation that reduces communication requirements.

99

Convex hull on the IBM SP2
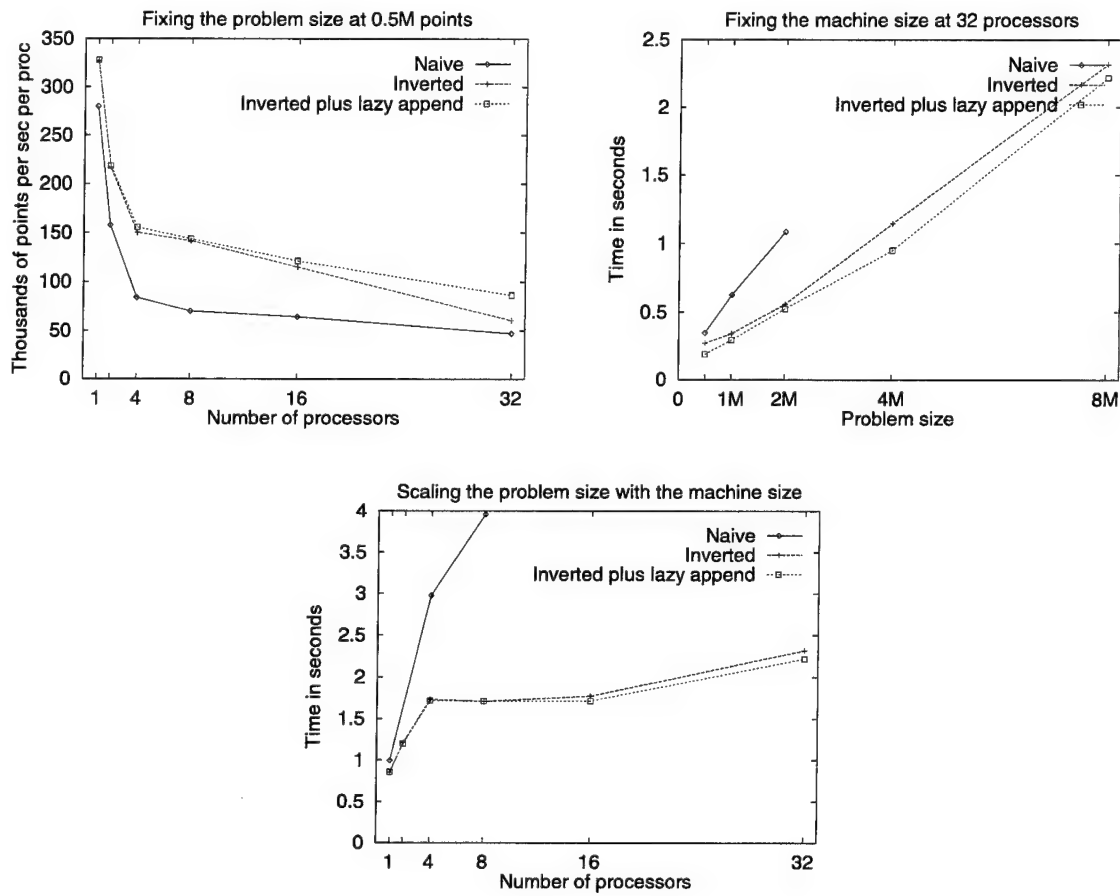
Figure 6.9: Three views of the performance of convex hull in Machiavelli on the IBM SP2. Views are as in Figure 6.8.

Convex hull on the SGI Power Challenge
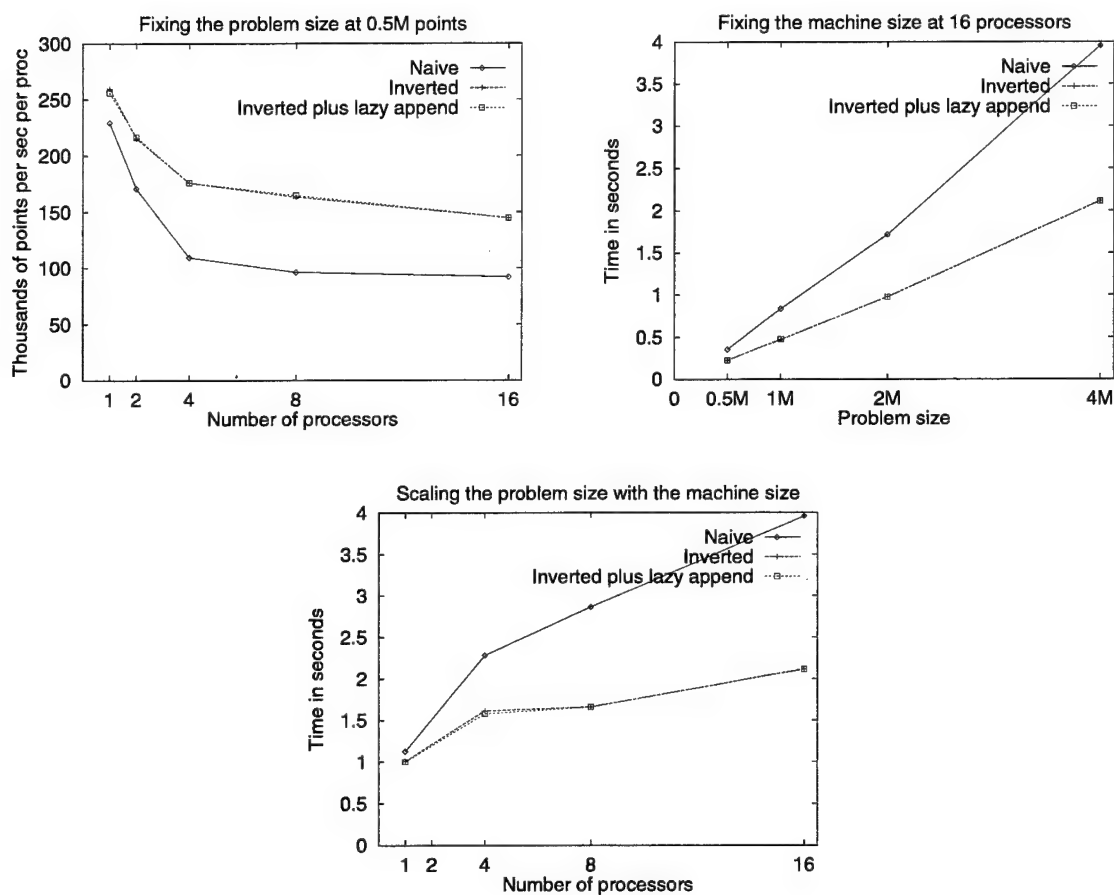

Figure 6.10: Three views of the performance of convex hull in Machiavelli on the SGI Power Challenge. Views are as in Figure 6.8. Note that the lazy append optimization, which reduces all-to-all communication, has very little effect on this shared memory architecture due to the ability to send messages by merely swapping pointers.

is varied. The lazy append gives little or no performance improvement on the SGI Power Challenge. This is due to the ability to "send" messages on this machine by merely swapping pointers into the shared memory address space, as noted in Section 5.2.2.

Apart from this effect, the shapes and relationships of the graphs on the different machines are again very similar. The use of the inverted `hsplit()` function improves overall performance by a factor of 1.5–2. In addition, it allows the solution of large problems that cannot be handled by the basic `hsplit()` function because of unnecessary vector replication. Overall, parallel efficiencies of 26–57% are achieved, even at small problem sizes.

6.3 Graph Separator

Figure 6.11 shows a geometric graph separator algorithm expressed in NESL, and Figure 6.12 shows the same algorithm in Machiavelli. This is the most complex of the algorithm kernels. The task is to separate an undirected graph into a number of equally-sized subgraphs, while minimizing the number of edges between subgraphs. The algorithm is a simple recursive bisection separator, which recursively chops its input into two equally-sized pieces along the x or y axes, choosing whichever separation gives the least number of cut edges on any particular level. The points are represented by a vector of point structures (tuples of floating-point numbers), and the edges by a vector of structures containing two integers, which hold the indices of the start and end points of each edge.

6.3.1 Finding a median

The first task in each recursive step of the separator is to choose an x or y separation. This is based on picking the median value of points along the x and y axes, and we therefore need a median-finding algorithm as a substep of the overall separator algorithm. NESL has an inbuilt `median()` function; Figure 6.13 shows an equivalent median algorithm written in Machiavelli. This separates the points into those less than and greater than a pivot value, and then chooses the appropriate half in which to recurse. However, note that it is singly recursive, and will thus be translated into purely data-parallel code by the Machiavelli preprocessor, with no use of control parallelism. Additionally, although the argument vector becomes shorter and shorter during the algorithm's progression, it will still be spread across the whole machine, and therefore more and more of the algorithm's time will be spent in parallel overhead (redistributing the vector on each recursive step) rather than in useful work (performing comparisons).

To overcome this problem, we can use a median-of-medians algorithm, in which each processor first finds the median of its local data, then contributes to a P-element vector in a collective communication step, and then finds the median element of those P medians. To do this

```
% Relabels node numbers, start at 0 in each partition %
function new_numbers(flags) =
let down = enumerate({not(flags): flags});
    up =    enumerate(flags);
in {select(flags;up,down): flags; up; down}

% Fetches from edges according to index vector v %
function get_from_edges(edges,v) =
{(e1,e2) : e1 in v->{e1: (e1,e2) in edges} ;
           e2 in v->{e2: (e1,e2) in edges} }

% Takes a graph and splits it into two graphs.
side   -- which side of partition point is on
points -- the coordinates of the points
edges  -- the endpoints of each edge
%
function split_graph(side,points,edges) =
let

% Relabel the node numbers %
new_n = new_numbers(side);

% Update edges to point to new node numbers %
new_edges = get_from_edges(edges,new_n);
new_side  = get_from_edges(edges,side);

% Edges between nodes that are both on one side %
eleft  = pack(zip(new_edges,{not(s1 or s2) :
                            (s1,s2) in new_side}));

eright = pack(zip(new_edges,{s1 and s2 :
                            (s1,s2) in new_side}));

% Back pointers to reconstruct original graph %
(sizes,sidx) = split_index(side)
in (split(points,side),vpair(eleft,eright),sidx)
```

```
% Checks how many edges are cut by a partition along
    specified coordinates %
function check_cut(coordinates,edges) =
let median = median(coordinates);
    flags = {x > median : x in coordinates};
    number = count({e1 xor e2 : (e1,e2) in
                    get_from_edges(edges,flags)});
in (number,flags)

% Returns the partition along one of the dimensions
  that minimizes the number of edges that are cut %
function find_best_cut(points,edges,dims_left) =
let dim = dims_left-1;
    (cut,flags) = check_cut({p[dim]:p in points},edges)
in if (dim == 0) then (cut,flags)
   else let (cut_r,flags_r) =
            find_best_cut(points,edges,dims_left-1)
        in if (cut_r < cut) then (cut_r,flags_r)
           else (cut,flags)

% The separator itself
dims  -- number of dimensions
depth -- depth in the recursion
count -- count on current level of recursion tree
%
function separator(dims,points,edges,depth,count) =
if (depth == 0) then dist(count,#points)
else
    let (cuts,side) = find_best_cut(points,edges,dims);
        (snodes,sedges,sindex) =
            split_graph(side,points,edges);
        result = {separator(dims,n,e,depth-1,c) :
                  n in snodes; e in sedges;
                  c in vpair(count*2,1+count*2)};
    in flatten(result)->sindex
```

Figure 6.11: **Two-dimensional geometric graph separator algorithm in NESL. It uses a simple recursive bisection technique, choosing the best axis along which to make a cut at each level.**

```
int crossing (vec_point left, vec_point right,
              double m, int ax)
{
  vec_int t;
  int result;

  t = {((l.coord[ax] < m) && (r.coord[ax] > m)) ||
       ((l.coord[ax] > m) && (r.coord[ax] < m))
      : l in left, r in right };
  result = reduce_sum (t);
  free_vec (t);
  return result;
}

void best_cut (vec_point p, vec_point left,
               vec_point right, double *p_m, int *p_ax)
{
  point median_x, median_y;
  int n_x, n_y;

  /* Find the median along each axis */
  median_x = median (p, 0);
  median_y = median (p, 1);

  /* Count edges cut by a partition on each median */
  n_x = crossing (left, right, median_x.coord[0], 0);
  n_y = crossing (left, right, median_y.coord[1], 1);

  /* Choose median and axis with fewer cuts */
  if (n_x < n_y) {
    *p_m = median_x.coord[0]; *p_ax = 0;
  } else {
    *p_m = median_y.coord[1]; *p_ax = 1;
  }
}


/* depth -- depth in the recursion
   count -- count on current level of recursion tree
*/
vec_point separator (vec_point pts, vec_edge edges,
                     int depth, int count)
{
  if (depth == 0) {
    result = { set_tag (p, count) : p in pts }
  } else {
    /* Fetch the points at the end of each edge */
    end_pts = fetch_pts (pts, edges);

    /* Separate them into even and odd points */
    even = even (end_pts, 1);
    odd = odd (end_pts, 1);

    /* Find the axis to separate on */
    best_cut (pts, even, odd, &m, &ax);

    /* Work out which side to pack things to */
    flags = { p.coord[ax] < m : p in pts }

    /* Pack the points to left and right. */
    l_pts = { p : p in pts, f in flags | f }
    r_pts = { p : p in pts, f in flags | !f }

    /* Pack all those edges which are in one side */
    l_edges = { e : e in edges, l in even, r in odd |
                ((l.coord[ax] < m) && (r.coord[ax] < m)) }
    r_edges = { e : e in edges, l in even, r in odd |
                ((l.coord[ax] > m) && (r.coord[ax] > m)) }

    /* Renumber edges */
    not_flags = { not(f) : f in flags }
    l_scan = scan_sum (flags);
    r_scan = scan_sum (not_flags);
    new_indices = { (f ? l : r) : f in flags,
                    l in l_scan, r in r_scan }
    l_edges = fetch_edges (new_indices, l_edges);
    r_edges = fetch_edges (new_indices, r_edges);

    split (left  = separator (l_pts, l_edges,
                              depth-1, 2*count),
           right = separator (r_pts, r_edges,
                              depth-1, 2*count+1));

    result = append (left, right);
  }
  return result;
}
```

Figure 6.12: Two-dimensional geometric graph separator algorithm in Machiavelli. Variable declarations and vector frees have been omitted from the separator function for clarity. Compare to the NESL code in Figure 6.11

```
typedef struct point {
  double coord[2];
}

point select_n (vec_point src, int k, int axis)
{
  vec_point les, grt;
  point pivot, result;
  int offset, index;

  index = length (src) / 2;
  pivot = get (src, index);
  les = { l : l in src | l.coord[axis] < pivot.coord[axis] };
  grt = { g : g in src | g.coord[axis] > pivot.coord[axis] };
  offset = length (src) - length (grt);
  free (src);

  if (k < length (les)) {
    free (grt);
    result = select_n (les, k, axis);
  } else if (k >= offset) {
    free (les);
    result = select_n (grt, k - offset, axis);
  } else {
    free (les); free_vec (grt);
    result = pivot;
  }
  return result;
}

point median (vec_point src, int axis)
{
  vec_point tmp = copy_vec (src);
  return select_n (src, length (src) / 2, aix);
}
```

Figure 6.13: Generalised n-dimensional selection algorithm in Machiavelli, together with a 2D point structure whose type it is specialized for. The basic selection algorithm finds the kth smallest element on the given axis; an additional wrapper function then uses it to find the median. Note that the algorithm is singly recursive.

```
point fast_median (vec_point src, int axis)
{
  point local;
  vec_point all;

  /* Find local median */
  local = select_n_serial (src, length (src) / 2, axis);

  /* Create a local vector of length P */
  all = alloc_vec_point_serial (tm->nproc);

  /* Gather the median from each processor into this vector */
  MPI_Allgather (&local, 1, _mpi_point,
                 all.data, 1, _mpi_point, tm->com);

  /* Return the median of the vector */
  return select_n_serial (all, length (all) / 2, axis);
}
```

Figure 6.14: Median-of-medians algorithm in Machiavelli. It uses the serial function se-lect_n_serial generated from the selection algorithm in Figure 6.13.

we can use the serial version of the algorithm in Figure 6.13 as our local median-finding algorithm, and exchange the elements using an MPI gather operation operating on the point type, as shown in Figure 6.14 (the gather operation could also be made into a Machiavelli primitive).

6.3.2 The rest of the algorithm

Having found two medians, the separator algorithm now counts the number of edges cut by the two separations and chooses the one creating the least number of cuts. It therefore fetches all the points using a global fetch operation, divides them into start and end points using even and odd, and iterates over the two sets, computing which pairs of points represent an edge crossed by the cut. We then sum the number of cuts, and pick the separation with the smaller sum. Packing the points and edges to the left and right of the separation is trivial. However, the edges are now numbered incorrectly, since their end-points were moved in the packing process. We correct this by performing a plus-scan on the flags used to pack the points and edges; this gives the new edge index for every point, which we can then fetch via the old edge index. This results in the full Machiavelli algorithm shown in Figure 6.12; the NESL algorithm in Figure 6.11 performs the same actions, but is structured slightly differently in terms of function layout.

Given that the division in this algorithm is chosen by finding a median value, it is inherently balanced, and load balancing would not affect its performance. Therefore, only two variants of the algorithm were tested, with and without the faster median substep.

106

Geometric graph separator on the Cray T3D
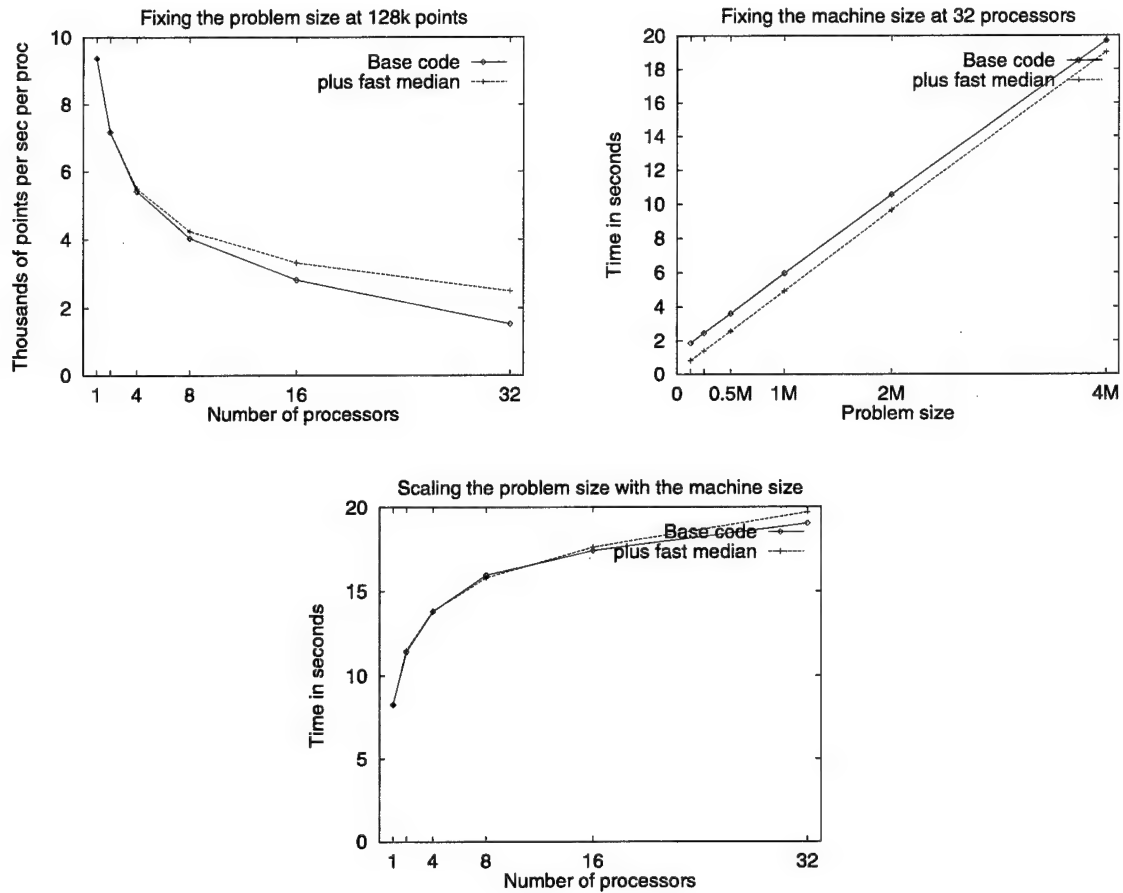
Figure 6.15: Three views of the performance of a geometric graph separator in Machiavelli on the Cray T3D. At the top left is the per-processor performance for a fixed problem size (0.125M points), as the problem size is varied. At the top right is the time taken for 32 processors, as the problem size is varied. At the bottom is the time taken as the problem size is scaled with the number of processors (from 0.125M points on 1 processor to 4M points on 32 processors. Times are shown for both the basic algorithm and a version using the faster median.

107

Figure 6.16: Three views of the performance of a geometric graph separator in Machiavelli on the IBM SP2. Views are as in Figure 6.15.
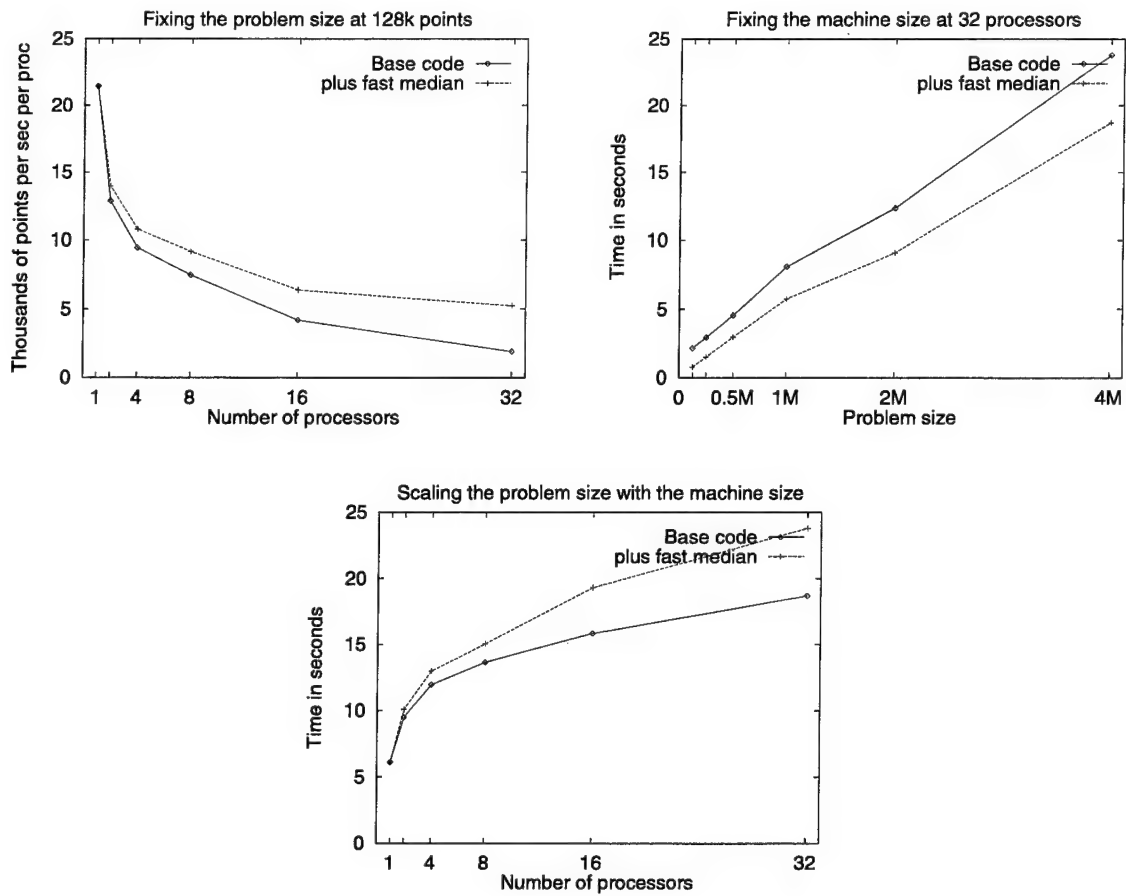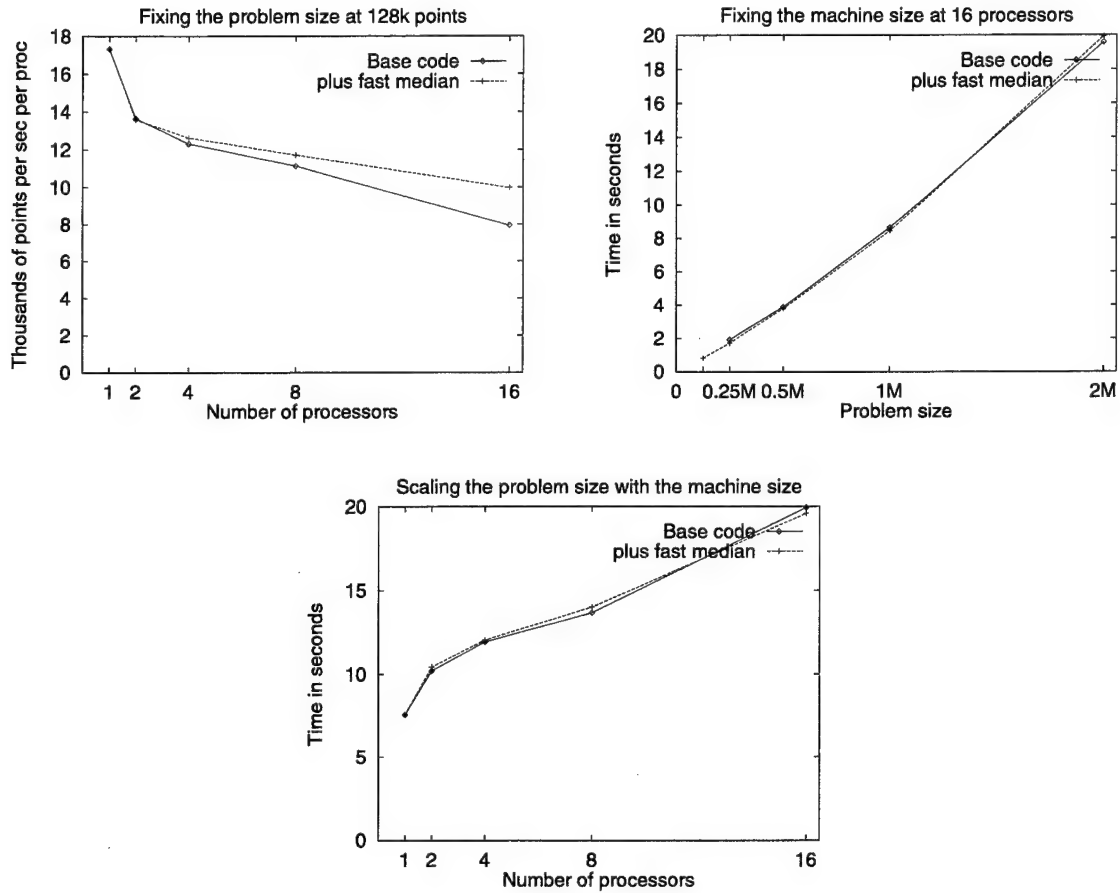
Geometric separator on the SGI Power Challenge

Figure 6.17: Three views of the performance of a geometric graph separator in Machiavelli on the SGI Power Challenge. Views are as in Figure 6.15.

109

6.3.3 Performance

Figures 6.15, 6.16, and 6.17 show the performance of the geometric graph separator algorithm on the Cray T3D, IBM SP2, and SGI Power Challenge, respectively. All times shown are to subdivide the initial graph into 32 equally-sized pieces. The graphs were Delaunay triangulations of points generated with a uniform distribution.

Once again, the shapes and relationships of the graphs are similar across the three machines. With the exception of the Power Challenge, the fast median algorithm improves performance by more than a factor of two for small problems on large numbers of processors. On the Power Challenge the redistribution of vectors on recursive steps of the original median algorithm is once again accomplished by simple pointer swapping between processors, and therefore the fast median algorithm has a relatively small effect at large problem sizes. The performance advantage of the fast median algorithm is also most apparent for large numbers of processors, where it eliminates many rounds of recursion and all-to-all communication.

Note that this algorithm scales significantly worse than quicksort and convex hull for a fixed problem size, but does well when the problem size is scaled with the machine size. The reason is the three global `fetch` operations on each recursive step, each of requires two rounds of all-to-all communication (one to send the indices of the requested elements, and a second to return the elements to the requesting processors). With more processors involved, this all-to-all communication takes longer, adding extra parallel overhead to the algorithm.

6.4 Matrix Multiplication

The final algorithm kernel is dense matrix multiplication, chosen as an example of an explicitly balanced algorithm that also involves significant computation, since in the naive case it performs $O(n^3)$ work. This is very easy to express in NESL, as shown in Figure 6.18; return to each element of the result the dot-product of a row times a column (that is, a transposed row). However, this algorithm also takes $O(n^3)$ space, and hence is impractical for large matrices.

```
function matrix_multiply(A,B) =
  {{sum({x*y: x in rowA; y in colB}) : colB in transpose(B)} : rowA in A}
```

Figure 6.18: Simple matrix multiplication algorithm in NESL

A more practical variant written using Machiavelli is shown in Figure 6.19. This divides the problem into two and recurses on each half. The base case performs an outer product on two vectors of length n to create an $n \times n$ array. This still imposes an upper limit on the maximum problem size that can be achieved in team-parallel mode, since each node must have sufficient

110

```
vec_double balanced mat_mul (vec_double A, vec_double B, int n)
{
  vec_double result;

  if (n == 1) {
    /* Base case is outer product of two vectors of length n */
    vec_double rep_A, rep_B, trans_A;
    rep_A = replicate (A, length (A));
    rep_B = replicate (B, length (B));
    trans_A = transpose (rep_B);
    result = { a * b : a in trans_A, b in rep_B };
    free(A); free(B); free(rep_A); free(rep_B); free(trans_A);
  } else {
    vec_double A0, A1, B0, B1, R0, R1;
    int half_n = n/2, half_size = length (A) / 2;

    /* Compute left and right halves of A, top and bottom of B */
    A0 = even (A, half_n);    A1 = odd (A, half_n);
    B0 = even (B, half_size); B1 = odd (B, half_size);
    free (A); free (B);

    /* Recurse two ways, computing two whole matrices */
    split (R0 = mat_mul (A0, B0, half_n),
           R1 = mat_mul (A1, B1, half_n));

    /* Add the resulting matrices together */
    result = { a + b : a in R0, b in R1 };
    free (R0); free (R1);
  }
  return result;
}
```

Figure 6.19: Two-way matrix multiplication algorithm in Machiavelli. The base case, when A and B are vectors of length n, performs an outer product of the vectors, creating a vector representing an array of size $n \times n$.

memory for an n^2 array, but for $n \gg p$ it allows significantly larger problems than assuming an n^3 array spread over p processors.

Figures 6.20, 6.21 and 6.22 show the performance of this two-way matrix multiplication algorithm on the Cray T3D, IBM SP2, and SGI Power Challenge, respectively. Note that the format of these graphs is different from those shown previously, to accomodate the $O(n^3)$ complexity of the algorithm. The upper two graphs on each page use a log scale on the time axis, and show two views of the same data, with all machine sizes and problem sizes plotted. The lower graph shows a classical speedup graph—speedup versus number of processors—for various problem sizes, with a perfect speedup line superimposed on the graph.

Note that the smaller problem sizes (e.g., 32×32) are much smaller than for previous algorithms, and cannot be effectively parallelized—adding processors makes them slower, since parallel overhead dominates the code. However, for large problem sizes the performance scales very well with machine size, and indeed scales better than any of the previous algorithms. This reflects both the higher ratio of computation to communication in this algorithm, and its inherent regularity, with no imbalance between processors. Slight superlinear effects on the T3D and SP2 are probably due to the larger amount of cache available to the algorithm as the number of processors is increased. Finally, note the catastrophic performance of large numbers of processors of the SGI Power Challenge on small problem sizes, due to access contention for the shared memory.

6.5 Summary

In this chapter I have presented Machiaveli benchmark results for four different algorithms: quicksort, convex hull, a geometric separator, and dense matrix multiplication. In addition, the benchmarks have been run on three different machines: an IBM SP2, an SGI Power Challenge, and a Cray T3D. These machines span the spectrum of processor/memory coupling, from the loosely-coupled distributed-memory SP2 to the tightly-coupled shared-memory Power Challenge.

Table 6.1 shows a summary of these results in terms of parallel efficiency (that is, percentage of perfect speedup over good serial code). The problem sizes are the largest that can be solved on a single processor of all the machines. Although this allows easy comparison between 1-processor and n-processor results, it results in artificially small problem sizes. As has been shown previously in this chapter, all of the machines exhibit better performance if problem size are scaled with the number of processors.

Note that the distributed-memory T3D and SP2 generally exhibit similar parallel efficiencies, while the shared-memory Power Challenge typically has a higher parallel efficiency. Some of this difference can be ascribed to the smaller number of processors in the Power

Matrix multiplication on the Cray T3D
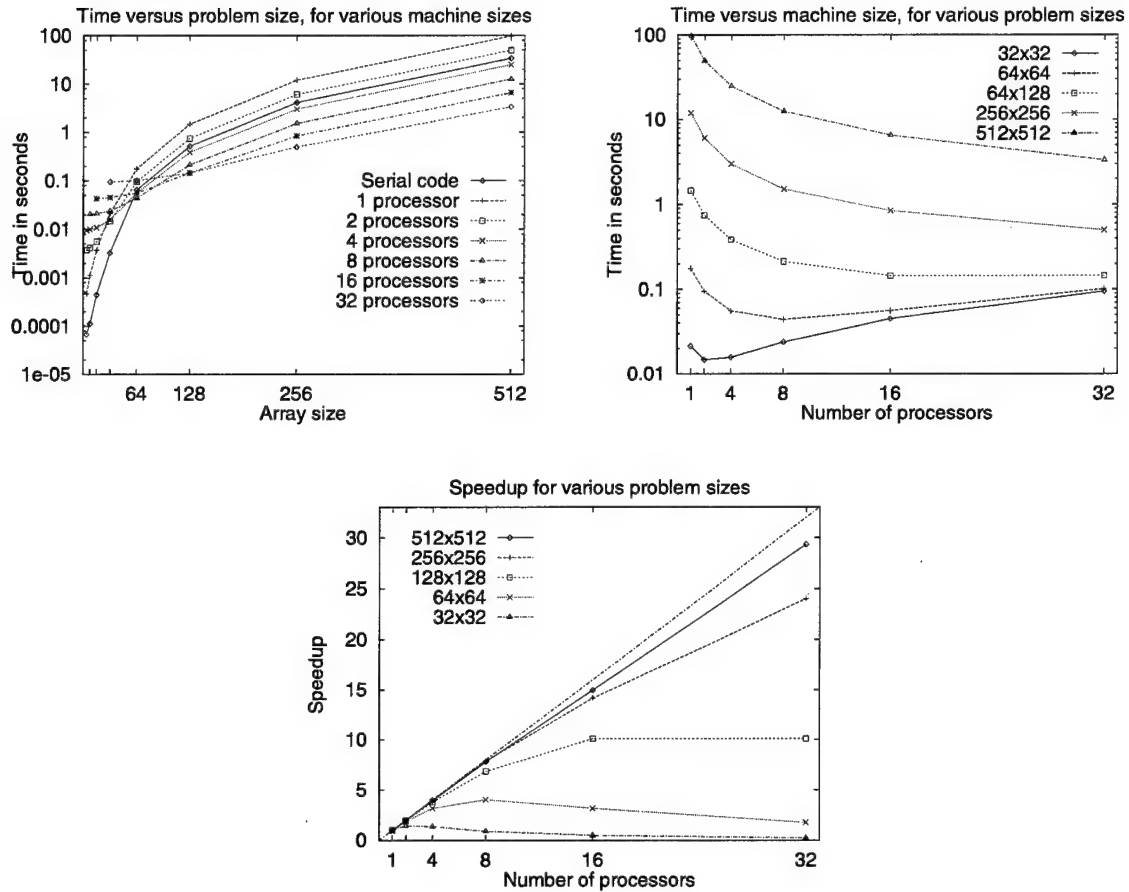
Figure 6.20: Three views of the performance of two-way matrix multiplication on the Cray T3D. This is an $O(n^3)$ algorithm, so the top two graphs use a log scale to show times. Specifically, at top left is the time taken versus the problem size, for a range of machine sizes. At the top right is the time taken versus the machine size, for a range of problem sizes. At the bottom is the speedup over single-processor code, for various problem sizes.

Matrix multiplication on the IBM SP2
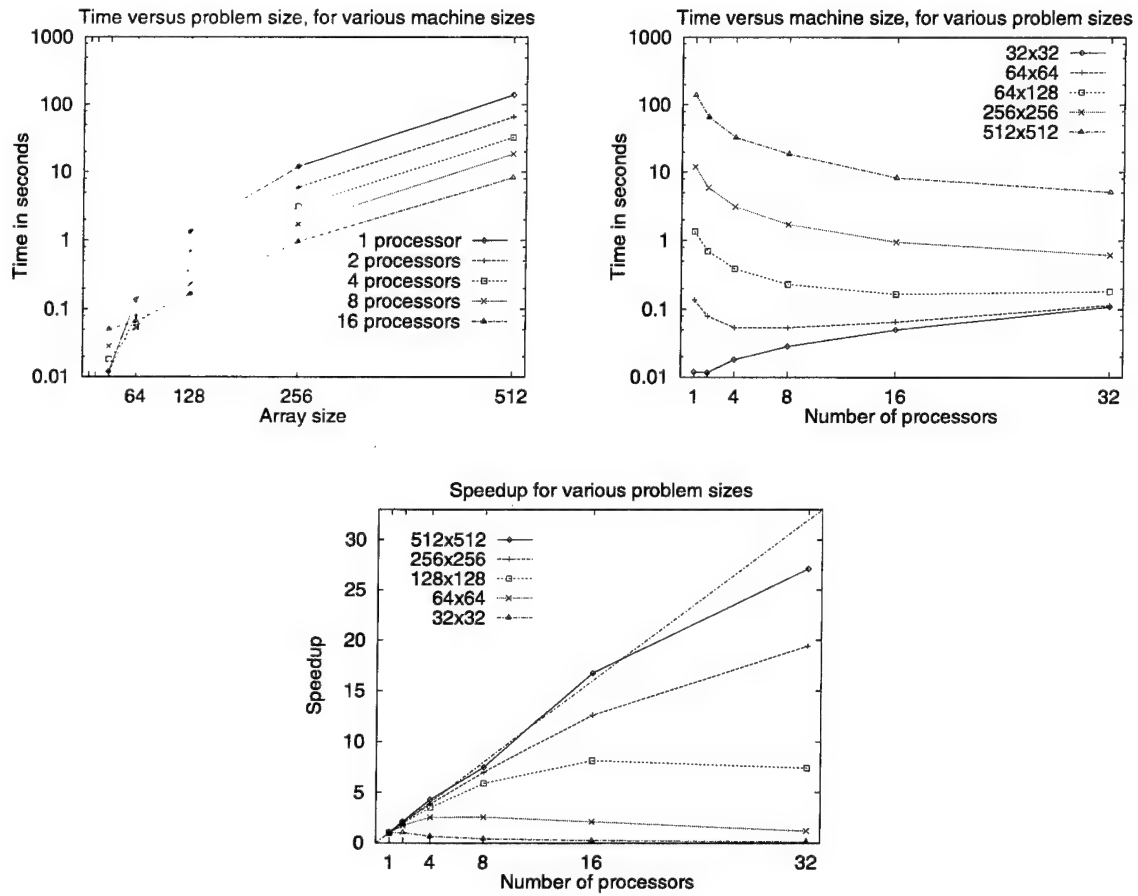
Figure 6.21: Three views of the performance of two-way matrix multiplication on the IBM SP2. Views are as in Figure 6.20.

114

Matrix multiplication on the SGI Power Challenge
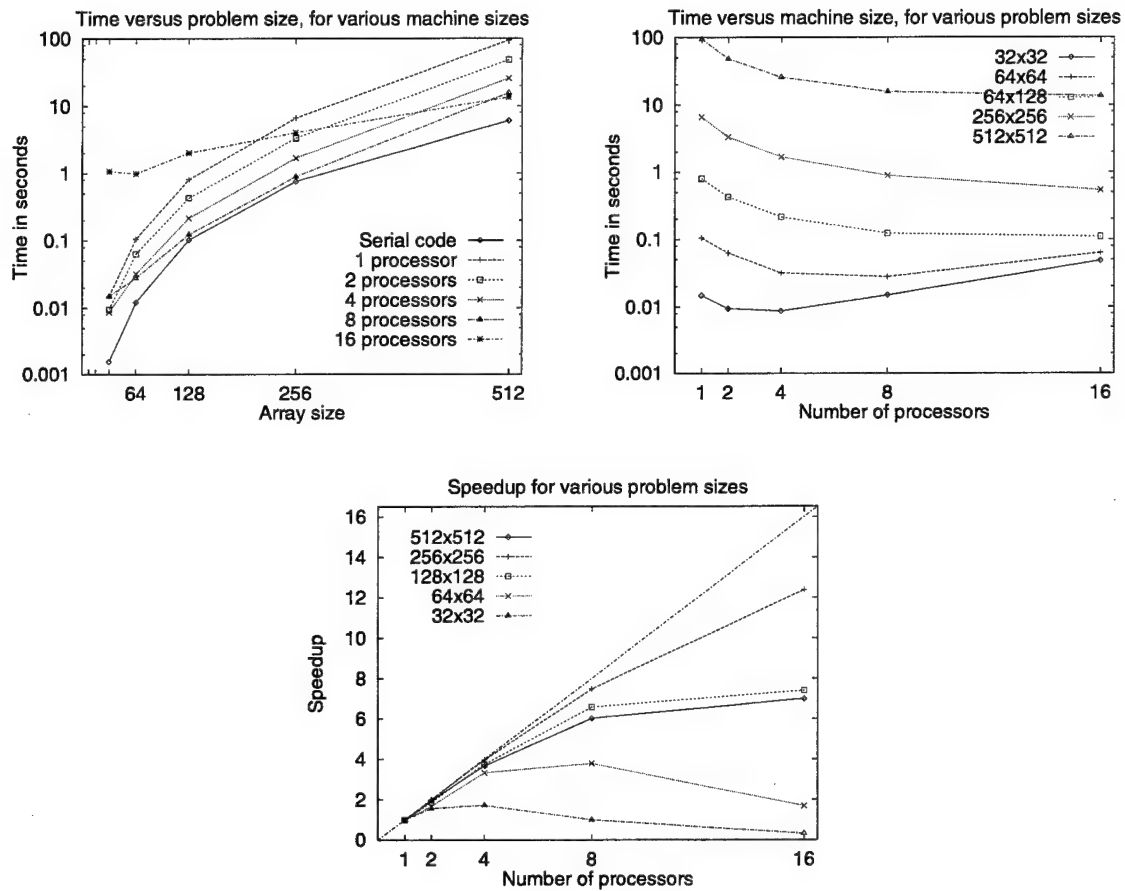
Figure 6.22: Three views of the performance of two-way matrix multiplication on the SGI Power Challenge. Views are as in Figure 6.20.

115

Algorithm	Problem size	T3D	SP2	SGI
Quicksort	2M numbers	30%	22%	
Convex hull	0.5M points	41%	26%	57%
Separator	0.125M points	27%	24%	58%
Matrix multipy	512x512	92%	85%	44%

Table 6.1: Parallel efficiencies (percentage of perfect speedup over serial code) of the algorithms tested in this chapter. Cray T3D and IBM SP2 results are for 32 processors, while the SGI Power Challenge results are for 16 processors.

Challenge, and some to its closer coupling of processors and memory. However, the Power Challenge does poorly on the matrix multiply benchmark precisely because of its shared memory, as processors contend for access to relatively small amounts of data. By comparison, the T3D and SP2 do much better on the matrix multiplication, since it has a high ratio of computation to communication.

Given that the first three benchmarks are for irregular algorithms with high communication requirements, and that the benchmarks are "worst case" in terms of the amount of data per processor, I believe that these parallel efficiencies demonstrate my thesis that irregular divide-and-conquer algorithms can be efficiently mapped onto distributed-memory machines.

Chapter 7

A Full Application:
Delaunay Triangulation

> *If you fit in L1 cache, you ain't supercomputing.*
> —Krste Asanovic (krste@icsi.berkeley.edu)

This chapter describes the implementation and performance of a full application, namely two-dimensional Delaunay triangulation, using an early version of the Machiavelli toolkit. It was developed before the preprocessor was available, but the structure of the code is similar, since the appropriate functions were generated by hand instead of with the preprocessor.

Although there have been many theoretical parallel algorithms for Delaunay triangulation, and a few parallel implementations specialized for uniform input distributions, there has been little work on implementations that work well for non-uniform distributions. The Machiavelli implementation uses the algorithm recently developed by Blelloch, Miller and Talmor [BMT96] as a coarse partitioner, switching to a sequential Delaunay triangulation algorithm on one processor. Specifically, I use the optimized version of Dwyer's algorithm [Dwy86] that is part of the Triangle mesh-generation package by Shewchuk [She96b]. As such, one goal of this chapter is demonstrate the reuse of existing efficient sequential code as part of a Machiavelli application.

A second goal is to show that Machiavelli can be used to develop applications that are both faster and more portable than previous implementations. The final Delaunay triangulation, after optimization of some sub-algorithms, is three times as efficient as the best previous parallel implementations, where efficiency is measured in terms of speedup over good sequential code. Additionally, it shows good scalability across a range of machine sizes and architectures, and (thanks to the original algorithm by Blelloch, Miller and Talmor) can handle non-uniform input distributions with relatively little effect on its performance. Overall, it is the fastest, most

117

portable, and most general parallel implementation of two-dimensional Delaunay triangulation that I know of.

The rest of this chapter is arranged as follows. In Section 7.1 I define the problem of Delaunay triangulation and discuss related work, concentrating on previous parallel implementations. In Section 7.2 I outline the algorithm by Blelloch, Miller and Talmor. In Section 7.3 I describe an implementation using Machiavelli. In Section 7.4 I provide performance results for the IBM SP2, SGI Power Challenge, Cray T3D, and DEC AlphaCluster, and analyse the performance of some algorithmic substeps. Finally, Section 7.5 summarizes the chapter. Some of this chapter previously appeared in [Har97] and [BHMT].

7.1 Delaunay Triangulation

A Delaunay triangulation in the point set R^2 is the unique triangulation of a set S of points such that there are no elements of S within the circumcircle of any triangle. Finding a Delaunay triangulation—or its dual, the Voronoi diagram—is an important problem in many domains, including pattern recognition [BG81], terrain modelling [DeF87], and mesh generation for the solution of partial differential equations [Wea92]. Delaunay triangulations and Voronoi diagrams are among the most widely-studied structures in computational geometry, and have appeared in many other fields under different names [Aur91], including *domains of action* in crystallography, *Wigner-Seitz zones* in metallurgy, *Thiessen polygons* in geography, and *Blum's transforms* in biology.

In many of these domains the triangulations step is a bottleneck in the overall computation, making it important to develop fast algorithms for its solution. As a consequence, there are many well-known sequential algorithms for Delaunay triangulation. The best have been extensively analyzed [For92, SD95], and implemented as general-purpose libraries [BDH96, She96b]. Since these algorithms are time and memory intensive, parallel implementations are important both for improved performance and to allow the solution of problems that are too large for sequential machines.

However, although many parallel algorithms for Delaunay triangulation have been described [ACG+88, RS89, CGD90, Guh94], practical parallel implementations have been slower to appear, and are mostly specialized for uniform distributions [Mer92, CLM+93, TSBP93, Su94]. One reason is that the dynamic nature of the problem can result in significant inter-processor communication. This is particularly problematic for non-uniform distributions. Performing key phases of the algorithm on a single processor (for example, serializing the merge step of a divide-and-conquer algorithm, as in [VWM95]) reduces this communication, but introduces a sequential bottleneck that severely limits scalability in terms of both parallel speedup and achievable problem size. The use of decomposition techniques such as bucketing [Mer92, CLM+93, TSBP93, Su94], or striping [DD89] can also reduce communication.

These techniques allow the algorithm to quickly partition points into one bucket (or stripe) per processor, and then use sequential techniques within the bucket. However, the algorithms rely on the input dataset having a uniform spatial distribution of points in order to avoid load imbalances between processors. Their performance on non-uniform distributions more characteristic of real-world problems is significantly worse than on uniform distributions. For example, the 3D algorithm by Teng et al [TSBP93] was up to 5 times slower on non-uniform datasets than on uniform ones (using a 32-processor CM-5), while the 3D algorithm by Cignoni et al [CLM+93] was up to 10 times slower (using a 128-processor nCUBE).

Even when considered on uniform distributions, the parallel speedups of these algorithm over efficient sequential code has not been good, since they are typically much more complex than their sequential counterparts. This added complexity results in low *parallel efficiency*; that is, they achieve only a small fraction of the perfect speedup over efficient sequential code running on one processor. For example, Su's 2D algorithm [Su94] achieved speedup factors of 3.5–5.5 on a 32-processor KSR-1, for a parallel efficiency of 11–17%, while Merriam's 3D algorithm [Mer92] achieved speedup factors of 6–20 on a 128-processor Intel Gamma, for a parallel efficiency of 5–16%. Both of these results were for uniform datasets. The 2D algorithm by Chew et al [CCS97] (which solves the more general problem of constrained Delaunay triangulation in a meshing algorithm) achieves speedup factors of 3 on an 8-processor SP2, for a parallel efficiency of 38%. However, this algorithm currently requires that the boundaries between processors be created by hand.

7.2   The Algorithm

Recently, Blelloch, Miller and Talmor have described a parallel two-dimensional Delaunay triangulation algorithm [BMT96], based on a divide-and-conquer approach, that promises to solve both the complexity problem and the distribution problem of previous algorithms. The algorithm is relatively simple, and experimentally was found to execute approximately twice as many floating-point operations as an efficient sequential algorithm. Additionally, the algorithm uses a "marriage before conquest" approach, similar to that of the sequential DeWall triangulator [CMPS93]. This reduces the merge step of the algorithm to a simple join. Previous parallel divide-and-conquer Delaunay triangulation algorithms have either serialized this step [VWM95], used complex and costly parallel merge operations [ACG+88, CGD90], or employed sampling approaches that result in an unacceptable expansion factor [RS89, Su94]. Finally, the new algorithm does not rely on bucketing, and can handle non-uniform datasets.

The algorithm is based loosely on the Edelsbrunner and Shi algorithm [ES91] for finding a 3D convex hull, using the well-known reduction of 2D Delaunay triangulation to computing a convex hull in 3D. However, in the specific case of Delaunay triangulation the points are on the surface of a sphere or paraboloid, and hence Blelloch et al were able to simplify the algorithm,

119

reducing the theoretical work from $O(n \log^2 n)$ on an EREW PRAM to $O(n \log n)$, which is optimal for Delaunay triangulation. The constants were also greatly reduced.

Figure 7.1 gives a pseudocode description of the algorithm, which uses a divide-and-conquer strategy. Each subproblem is determined by a region $\mathcal{R}$ which is the union of a collection of Delaunay triangles. This region is represented by the polygonal border B of the region, composed of Delaunay edges, and the set of points P of the region, composed of *internal points* and points on the border. Note that the region may be unconnected. At each recursive call, the region is divided into two using a median line cut of the internal points. The set of internal points is subdivided into those to the left and to the right of the median line. The polygonal border is subdivided using a new path of Delaunay edges that corresponds to the median line: the new path separates Delaunay triangles whose circumcenter is to the left of the median line, from those whose circumcenter is to the right of the median line. Once the new path is found, the new border of Delaunay edges for each subproblem is determined by merging the old border with the new path, in the BORDER_MERGE subroutine. Some of the internal points may appear in the new path, and may become border points of the new subproblems. Since it is using a median cut, the algorithm guarantees that the number of internal points is reduced by a factor of at least two at each call.

The new separating path of Delaunay edges is a lower convex hull of a simple transformation of the current point set. To obtain this path $\mathcal{H}$ we project the points onto a paraboloid whose center is on the median line L, then project the points horizontally onto a vertical plane whose intersection with the x-y plane is L (see Figure 7.2). The two dimensional lower convex hull of those projected points is the required new border path $\mathcal{H}$. This divide-and-conquer method can proceed as long as the subproblem contains internal points. Once the subproblem has no more internal points, it is a set of (possibly pinched) cycles of Delaunay edges. There may be some missing Delaunay edges between border points that still have to be found. To do that, the algorithm moves to the END_GAME.

7.2.1 Predicted performance

The performance of the Delaunay triangulation algorithm depends on the complexity of the three subroutines, which can be chosen as follows:

LOWER_CONVEX_HULL: Since the projections are always on a plane perpendicular to the x or y axes, the points can be kept sorted relative to these axes with linear work. This in turn allows the use of the $O(n)$ work divide-and-conquer algorithm for 2D convex hull by Overmars and Van Leeuwens [OvL81].

BORDER_MERGE: This takes the old border B and merges it with the newly-found dividing path $\mathcal{H}$ to form a new border for a recursive call. The new border is computed based

---

Algorithm: DELAUNAY(P, B)

Input: P, a set of points in R^2,
B, a set of Delaunay edges of P which is the border of a region in R^2 containing P.

Output: The set of Delaunay triangles of P which are contained within B.

Method:

1. If all the points in P are on the boundary B, return END_GAME(B).

2. Find the point q that is the median along the x axis of all internal points (points in P and not on the boundary). Let $\mathcal{L}$ be the line $x = q_x$.

3. Let $P' = \{(p_y - q_y, ||p - q||^2) \mid (p_x, p_y) \in P\}$. These points are derived from projecting the points P onto a 3D paraboloid centered at q, and then projecting them onto the vertical plane through the line $\mathcal{L}$.

4. Let $\mathcal{H} = $ LOWER_CONVEX_HULL(P'). $\mathcal{H}$ is a path of Delaunay edges of the set P. Let $P_{\mathcal{H}}$ be the set of points the path $\mathcal{H}$ consists of, and $\bar{\mathcal{H}}$ be the path $\mathcal{H}$ traversed in the opposite direction.

5. Create two subproblems:

   - $B^L = $ BORDER_MERGE$(B, \mathcal{H})$
     $B^R = $ BORDER_MERGE$(B, \bar{\mathcal{H}})$

   - $P^L = \{p \in P | p \text{ is left of } \mathcal{L}\} \cup \{p' \in P_{\mathcal{H}} \mid p' \text{ contributed to } B^L\}$
     $P^R = \{p \in P | p \text{ is right of } \mathcal{L}\} \cup \{p' \in P_{\mathcal{H}} \mid p' \text{ contributed to } B^R\}$

6. Return DELAUNAY$(P^L, B^L) \cup $ DELAUNAY(P^R, B^R)

Figure 7.1: Pseudocode for the parallel Delaunay triangulation algorithm, taken from [BMT96] and correcting an error in step 5. Initially B is the convex hull of P. For simplicity, only cuts on the x axis are shown—the full algorithm switches between x and y cuts on every level. The three subroutines END_GAME, LOWER_CONVEX_HULL, and BORDER_MERGE are described in the text.

(a) The median line $\mathcal{L}$ and the path $\mathcal{H}$. (b) The points projected onto the paraboloid. (c) Paraboloid points projected onto the plane.
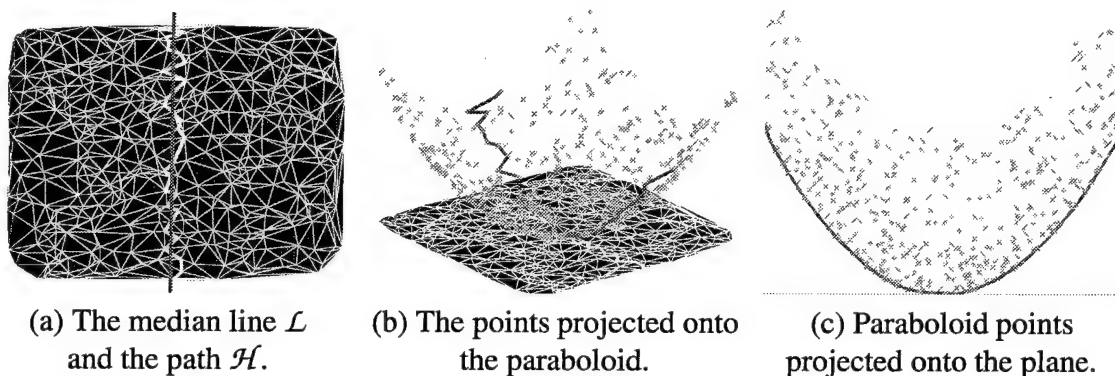
Figure 7.2: Finding a dividing path for Delaunay triangulation, taken from [BMT96]. This figure shows the median line, all the points projected onto a parabola centered at a point on that line, and the horizontal projection onto the vertical plane through the median line. The result of the lower convex hull in the projected space, $\mathcal{H}$, is shown in highlighted edges on the plane.

on an inspection of the possible intersections of points in B and $\mathcal{H}$. The intersection is based only on local structure, and can be computed in $O(n)$ work and $O(1)$ time.

END_GAME: When this stage is reached, the subproblems have no internal points, but may consist of several cycles. We proceed by finding a new Delaunay edge that splits a cycle into two, and then recursing on the two new cycles. A Delaunay edge can be found in $O(n)$ work and $O(1)$ time by using the duality of the set of Delaunay neighbors of a point q with the set of points on a 2D convex hull after inverting the points around q.

Blelloch et al found experimentally that a simple quickhull [PS85] was faster than a more complicated convex hull algorithm that was guaranteed to take linear time. Furthermore, using a point-pruning version of quickhull that limits possible imbalances between recursive calls [CSY95] reduces its sensitivity to non-uniform datasets.

With these changes, the parallel Delaunay triangulation algorithm was found to perform about twice as many floating-point operations as Dwyer's algorithm [Dwy86], which has been shown experimentally to be the fastest of the sequential algorithms [She96b, SD95]. This work efficiency of around 50% was maintained across a range of input distributions, including two with non-uniform distributions of points. Furthermore, the cumulative floating-point operation count was found to increase uniformly with recursion depth, indicating that the algorithm should be usable as a partitioner without loss of efficiency.

However, when implemented in NESL the algorithm was an order of magnitude slower on one processor than a good sequential algorithm, due to the performance limitations of NESL described in Chapter 2. The algorithm was chosen as a test case for Machiavelli due to its

recursive divide-and-conquer nature, the natural match of the partitioning variant to Machiavelli's ability to use efficient serial code, and its nesting of a recursive convex hull algorithm within a recursive Delaunay triangulation algorithm, as shown in Figure 1.3 on page 6.

7.3 Implementation in Machiavelli

As implemented in Machiavelli, the parallel algorithm is used only as a coarse partitioner, subdividing the problem into pieces small enough to be solved on a single processor using the sequential Triangle package [She96b]. This section describes several implementation decisions and optimizations that affect the performance of the final program, including choosing the data structures, improving the performance of specific algorithmic substeps, and using a "lazy append" optimization. Most of the optimizations reduce or eliminate interprocessor communication. Further analysis can be found in Section 7.4.

Data structures   The basic data structure used by the code is a point, represented using two double-precision floating-point values for the x and y coordinates, and two integers, one serving as a unique global identifier and the other as a communication index within team phases of the algorithm. The points are stored in balanced Machiavelli vectors. To describe the relationship between points in a border, the code uses *corners*. A corner is a triplet of points corresponding to two segments in a path. Corners are not balanced across the processors as points are, but rather are stored in unbalanced vectors on the same processor as their "middle" point. The other two points are replicated in the corner structure, removing the need for a global communication step when operating on them (see Section 7.4.1 for an analysis of the memory cost of the replication). In particular, the structure of corners and their unbalanced replication allows the complicated border merge step to operate on purely local data on each processor. Finally, an additional vector of indices I (effectively, pointers) links the points in P with the corners in the borders B and H. Given these data structures, I'll describe the implementation and optimization of each of the phases of the algorithm in turn.

Identifying internal points: Finding local internal points (those in P but not on the boundary B) is accomplished with a simple data-parallel operation across P and I that identifies points with no corresponding corner in B. These local points are rebalanced across the current team to create a new vector of points, using a single call to `pack`.

Finding the median: Initially a parallel version of the quickmedian algorithm [Hoa61] was used to find the median of the internal points along the x or y axis. This algorithm is singly-recursive, redistributing a subset of the data amongst the processors on each step, which results

123

in a high communication overhead. As in the separator algorithm of Chapter 6, it proved to be significantly faster to replace this with a median-of-medians algorithm, in which each processor first uses a serial quickmedian to compute the median of its local data, shares this local median with the other processors in a collective communication step, and finally computes the median of all the local medians. The result is not guaranteed to be the exact median, but in practice it is sufficiently good for load-balancing purposes. Although it is possible to construct input sets that would cause pathological behavior because of this modification, a simple randomization of the input data before use makes this highly unlikely in practice. Overall, the modification increased the speed of the Delaunay triangulation algorithm for the data sets and machine sizes studied (see Section 7.4) by 4–30%.

Project onto a parabola: Again, this is a purely local step, involving a simple data-parallel operation on each processor to create the new vector of points.

Finding the lower convex hull: The subtask of finding the lower 2D convex hull of the projected inner points of the problem was shown by Blelloch et al to be the major source of floating-point operations within the original algorithm, and it is therefore worthy of serious study. For the Machiavelli implementation, as in the original algorithm, a simple version of quickhull [PS85] was originally used. Quickhull is itself divide-and-conquer in nature, and is implemented as such using recursive calls to the Machiavelli toolkit, as shown in Section 6.2.

The basic quickhull algorithm is fast on uniform point distributions, but tends to pick extreme "pivot" points when operating on very non-uniform point distributions, resulting in a poor division of data and a consequent lack of progress. Chan et al [CSY95] describe a variant that uses the pairing and pruning of points to guarantee that recursive calls have at most 3/4 of the original points. Experimentally, pairing a random selection of $\sqrt{n}$ points was found to give better performance when used as a substep of the Delaunay triangulation algorithm than pairing all n points (see Section 7.4.2 for an analysis). As with the median-of-medians approach and the use of quickhull itself, the effects of receiving non-optimal results from an algorithm substep are more than offset by the decrease in running time of the substep. The result of this step is a vector of indices of the convex hull of the projected points. Calls to `fetch` are then used to fetch the points themselves, and the corners that they "own".

Combining hulls: As in Section 6.2, intermediate results of the quickhull function are left in an unbalanced state on the way back up the recursive call tree, and are handled with a single call to `pack` at the top level. This optimization eliminates $\log P$ levels of all-to-all communication.

Create the subproblems: Having found the hull and hence the dividing path $\mathcal{H}$, we can now merge the current border B with $\mathcal{H}$, creating two new borders B^L and B^R and partitioning

124

the points into P^L and P^R. Note that the merge phase is quite complicated, requiring line orientation tests to decide how to merge corners, but it is purely local thanks to the replicated corner structures. Although Figure 7.1 shows two calls to the border merge function (one for each direction of the new dividing path), in practice it is faster to make a single pass, creating both new borders and point sets at the same time.

End game: Since we are using the parallel algorithm as a coarse partitioner, the end game is replaced with a serial Delaunay triangulation algorithm. I chose to use the version of Dwyer's algorithm that is implemented in the Triangle mesh generation package by Shewchuk [She96b], which has performance comparable to that of the original code by Dwyer. Since the input format for Triangle differs from that used by the Machiavelli code, conversion steps are necessary before and after calling it. These translate between the pointer-based format of Triangle, which is optimized for sequential code, and the indexed format with triplet replication used by the parallel code. No changes are necessary to the source code of Triangle.

7.4   Experimental Results

The goal of this section is to validate my claims of portability, absolute efficiency compared to good sequential code, and ability to handle non-uniform input distributions. To test portability, I used four parallel architectures: the IBM SP2, SGI Power Challenge, and Cray T3D described in previous chapters, and a DEC AlphaCluster, which is a loosely-coupled workstation cluster with eight processors connected by an FDDI switch. The code on the AlphaCluster was compiled with `cc -O2` and MPICH 1.0.12. To test parallel efficiency, I compared timings on multiple processors to those on one processor, when the algorithm immediately switches to sequential Triangle code [She96b]. To test the ability to handle non-uniform input distributions I used four different distributions taken from [BMT96], and shown in Figure 7.3. They have the following characteristics:

Uniform distribution: Points are chosen at random within the unit square.

Normal distribution: The coordinates x and y are chosen as independent samples from the normal distribution.

Kuzmin distribution: This is an example of convergence to a point, and is used by astrophysicists to model the distribution of star clusters within galaxies [Too63]. It is radially symmetric, with density falling rapidly with distance r from a central point. The accumulative probability function is

$$M(r) = 1 - \frac{1}{\sqrt{1 + r^2}}$$

125

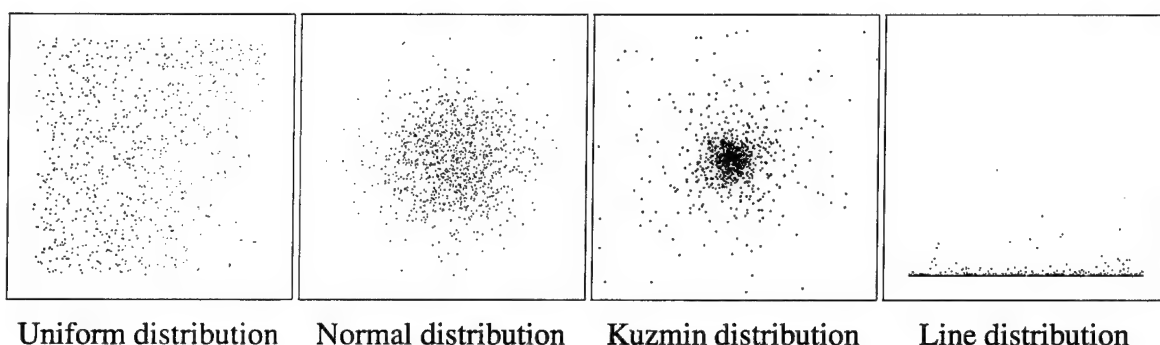| Uniform distribution | Normal distribution | Kuzmin distribution | Line distribution |

Figure 7.3: Examples of 1000 points in each of the four test distributions, taken from [BMT96]. For clarity, the Kuzmin distribution is shown zoomed on the central point. Parallelization techniques that assume uniform distributions, such as bucketing, suffer from poor performance on the Kuzmin and line distributions.

Line singularity: This is an example of convergence to a line, resulting in a distribution that cannot be efficiently parallelized using techniques such as bucketing. It is defined using a constant b (set to 0.001) and a transformation from the uniform distribution (u, v) of

$$(x, y) = \left(\frac{b}{u - bu + b}, v \right)$$

All timings represent the average of five runs using different seeds for a pseudo-random number generator. For a given problem size and seed the input data is the same regardless of the architecture and number of processors.

To illustrate the algorithm's parallel efficiency, Figure 7.4 shows the time to triangulate 128k points on different numbers of processors, for each of the four platforms and the four different distributions. This is the largest number of points that can be triangulated on one processor of all four platforms. The single-processor times correspond to sequential Triangle code. Speedup is not perfect because as more processors are added, more levels of recursion are spent in parallel code rather than in the faster sequential code. The Kuzmin and line distributions show similar speedups to the uniform and normal distributions, suggesting that the algorithm is effective at handling non-uniform datasets as well as uniform ones. Note that the Cray T3D and the DEC AlphaCluster use the same 150MHz Alpha 21064 processors, and their single-processor times are thus comparable. However, the T3D's specialized interconnection network has lower latency and higher bandwidth than the commodity FDDI network on the AlphaCluster, resulting in better scalability.

To illustrate scalability, Figure 7.5 shows the time to triangulate a variety of problem sizes on different numbers of processors. For clarity, only the uniform and line distributions are shown, since these take the least and most time, respectively. Again, per-processor performance

126

Figure 7.4: Speedup of Delaunay triangulation program for four input distributions and four parallel architectures. The graphs show the time to triangulate a total of 128k points as the number of processors is varied. Single processor results are for efficient serial code. Increasing the number of processors results in more levels of recursion being spent in slower parallel code rather than faster serial code, and hence the speedup is not linear. The effect of starting with an x or y cut on the highly directional line distribution is shown in performance that is alternately poor (at 1 and 4 processors) and good (at 2 and 8 processors).
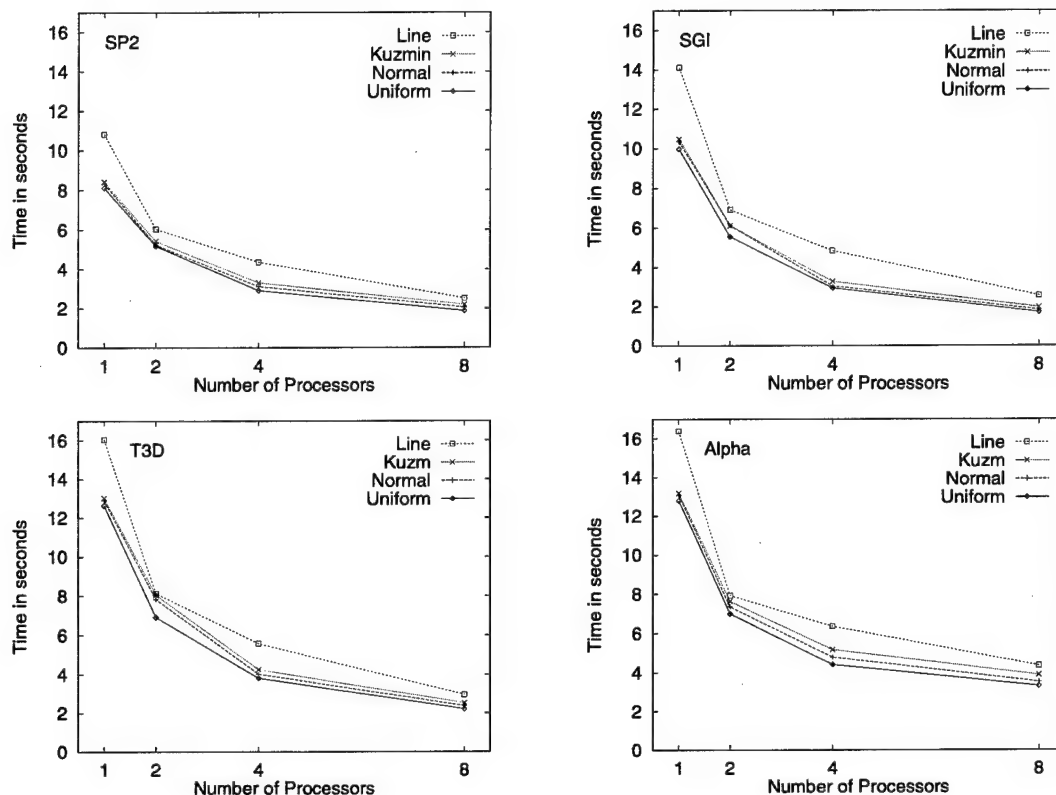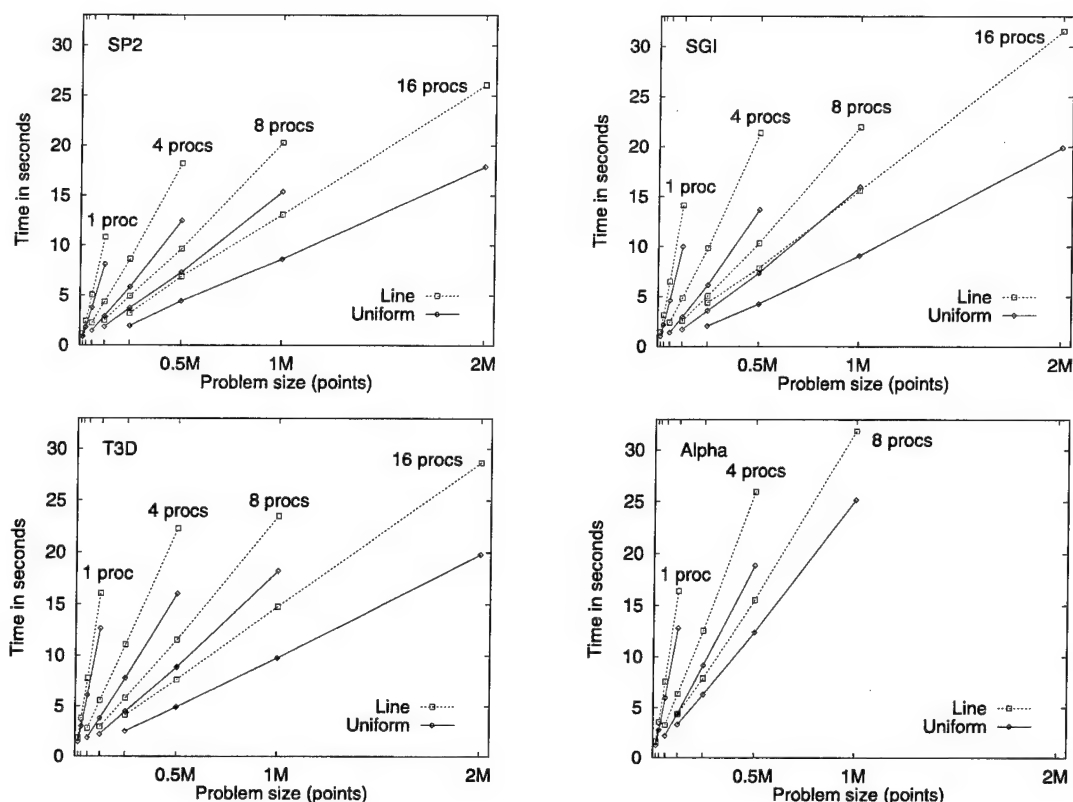
127

Figure 7.5: Scalability of Delaunay triangulation program for two input distributions and four parallel architectures. The graphs show the time to triangulate 16k-128k points per processor as the number of processors is varied. For clarity, only the fastest (uniform) and slowest (line) distributions are shown.

degrades as we increase the number of processors because more levels of recursion are spent in parallel code. However, for a fixed number of processors the performance scales very well with problem size.

To illustrate the relative costs of the different substeps of the algorithm, Figure 7.6a shows the accumulated time per substep. The parallel substeps, namely median, convex hull, and splitting and forming teams, become more important as the number of processors is increased. The time taken to convert to and from Triangle's data format is insignificant by comparison, as is the time spent in the complicated but purely local border merge step. Figure 7.6b shows the same data from a different view, as the total time per recursive level of the algorithm. This clearly shows the effect of the extra parallel phases as the number of processors is increased.

Finally, Figure 7.7 uses a parallel time line to show the activity of each processor when triangulating a line singularity distribution. There are several important effects that can be seen
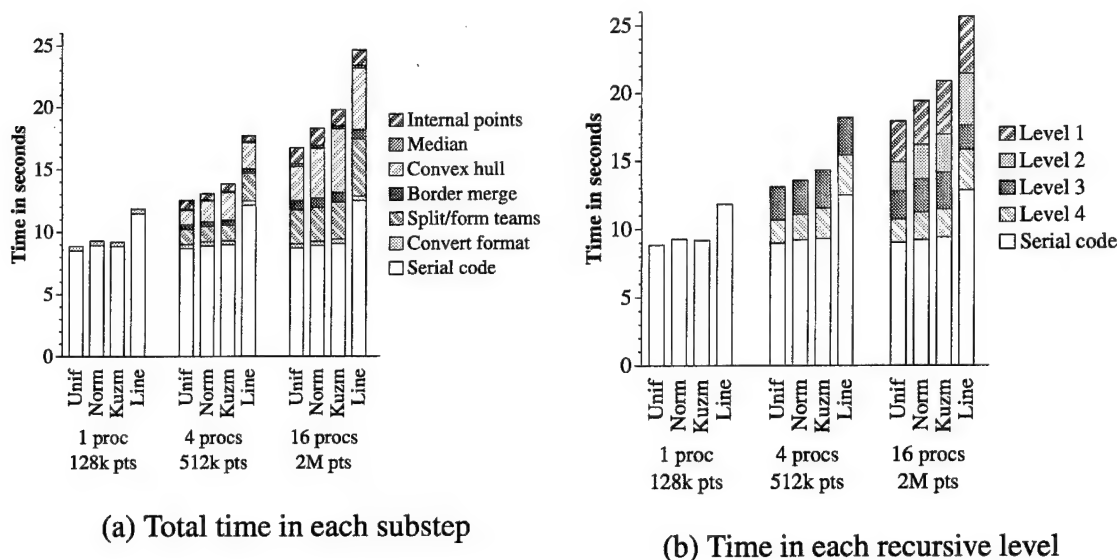
(a) Total time in each substep

(b) Time in each recursive level

Figure 7.6: Breakdown of time per substep of Delaunay triangulationm showing two views of the execution time as the problem size is scaled with number of processors (IBM SP2, 128k points per processor). (a) shows the total time spent in each substep of the algorithm; the time in sequential code remains approximately constant, while convex hull and team operations (which includes synchronization delays) are the major overheads in the parallel code. (b) shows the time per recursive level of the algorithm; note the additional overhead per level.

here. First, the nested recursion of the convex hull algorithm within the Delaunay triangulation algorithm. Second, the alternating high and low time spent in the convex hull, due to the effect of the alternating x and y cuts on the highly directional line distribution. Third, the operation of the processor teams. For example, two teams of four processors split into four teams of two just before the 0.94 second mark, and further subdivide into eight teams of one processor (and hence switch to sequential code) just after. Lastly, the amount of time wasted waiting for the slowest processor in the parallel merge phase at the end of the algorithm is relatively small, despite the very non-uniform distribution.

7.4.1 Cost of replicating border points

As explained in Section 7.3, border points are replicated in corner structures to eliminate the need for global communication in the border merge step. Since one reason for using a parallel computer is to be able to handle larger problems, we would like to know that this replication does not significantly increase the memory requirements of the program. Assuming 64-bit doubles and 32-bit integers, a point (two doubles and two integers) and associated index vector entry (two integers) occupies 32 bytes, while a corner (two replicated points) occupies 48
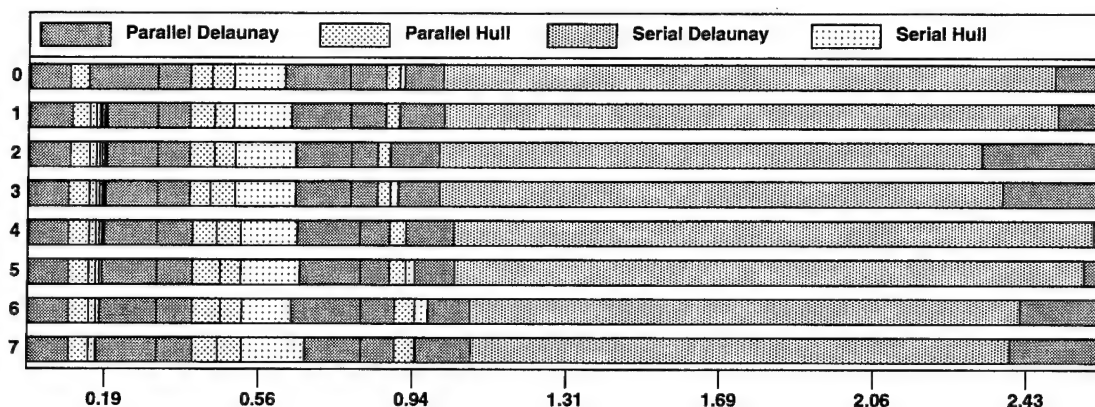
129

Figure 7.7: Activity of eight processors during Delaunay triangulation, showing the parallel and sequential phases, and the inner convex hull algorithm (IBM SP2, 128k points in a line singularity distribution). A parallel step consists of two phases of Delaunay triangulation code surrounding one or more convex hull phases; this run has three parallel steps. Despite the non-uniform distribution the processors do approximately the same amount of sequential work.

bytes. However, since a border is normally composed of only a small fraction of the total number of points, the additional memory required to hold the corners is relatively small. For example, in a run of 512k points in a line distribution on eight processors, the maximum ratio of corners to total points on a processor (which occurs at the switch between parallel and serial code) is approximately 2,000 to 67,000, so that the corners occupy less than 5% of required storage. Extreme cases can be manufactured by reducing the number of points per processor; for example, with 128k points the maximum ratio is approximately 2,000 to 17,500. Even here, however the corners still represent less than 15% of required storage, and by reducing the number of points per processor we have also reduced absolute memory requirements.

7.4.2 Effect of convex hull variants

As discussed in Section 7.3, the convex hull substep can have a significant effect on the overall performance of the Delaunay triangulation algorithm. To explore this, a basic quickhull algorithm was benchmarked against two variants of the pruning quickhull by Chan et al [CSY95]: one that pairs all n points, and one that pairs a random sample of $\sqrt{n}$ points. Results for an extreme case are shown in Figure 7.8. As can be seen, the n-pairing algorithm is more than twice as fast as the basic quickhull on the non-uniform Kuzmin distribution (over all the distributions and machine sizes tested it was 1.03–2.83 times faster). The $\sqrt{n}$-pairing algorithm provides a modest additional improvement, being 1.02–1.30 times faster still.
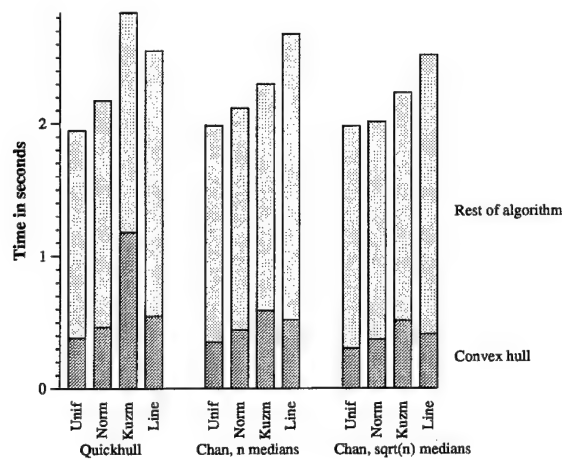
130

Figure 7.8: Effect of different convex hull functions on total time of Delaunay triangulation (128k points on an 8-processor IBM SP2). The pruning quickhull due to Chan et al [CSY95] has much better performance than the basic algorithm on the non-uniform Kuzmin distribution; using a variant with reduced sampling accuracy produces a modest additional improvement.

7.5   Summary

This chapter has described the use of the Machiavelli toolkit to produce a fast and practical parallel two-dimensional Delaunay triangulation algorithm. The code was derived from a combination of a theoretically efficient CREW PRAM parallel algorithm and existing optimized serial code. The resulting program has three advantages over existing work. First, it is widely portable due to its use of MPI; it achieves similar speedups on four machines with very different communication architectures. Second, it can handle datasets that do not have a uniform distribution of points with a relatively small impact on performance. Specifically, it is at most 1.5 times slower on non-uniform datasets than on uniform datasets, whereas previous implementations have been up to 10 times slower. Finally, it has good absolute performance, achieving a parallel efficiency (that is, percentage of perfect speedup over good serial code) of greater than 50% on machine sizes of interest. Previous implementations have achieved at most 17% parallel efficiency.

In addition to these specific results, there are two more general conclusions that are well-known but bear repeating. First, constants matter: simple algorithms are often faster than more complex algorithms that have a lower complexity (as shown in the case of quickhull). Second, quick but approximate algorithmic substeps often result in better overall performance than slower approaches that make guarantees about the quality of their results (as shown in the case of the median-of-medians and $\sqrt{n}$-pair pruning quickhull).

131

Chapter 8

Conclusions

I have seen the chapters of your work, and I like it very much;
but until I have seen the rest, I do not wish to express a definite judgement.
—Francesco Vettori, in a letter to Niccolò Machiavelli, 18 January 1514

In this dissertation I have presented a model and a language for implementing irregular divide-and-conquer algorithms on parallel architectures. This chapter summarizes the contributions of the work, and suggests some extensions and future work.

8.1 Contributions of this Work

The contributions of this thesis can be separated into the model, the language and system, and the results. I will tackle each of these in turn.

8.1.1 The model

I have described the team-parallel model, which is targetted at irregular divide-and-conquer algorithms. The team-parallel model achieves nested parallelism by using data-parallel operations within teams of processors that are themselves acting in a control-parallel manner. It assumes that data is stored in collection-oriented data types, such as vectors or sets, that can be manipulated using data-parallel operations. Recursion in the algorithm is mapped naturally to the subdivision of processor teams. When a team has subdivided to the point where it only contains a single processor, it switches to efficient serial code, which has no parallel overheads and can be replaced by specialized user-supplied code where available. Finally, a load-balancing system is used to compensate for the imbalances introduced by irregular algorithms. All of

these concepts have previously been implemented for divide-and-conquer algorithms, but the team model is the first to combine all four.

I have also defined seven axes along which to classify divide-and-conquer algorithms: branching factor, balance, embarassing divisibility, data dependency in the divide function, data dependency in the output size, control parallelism, and data parallelism. Using these axes I have described a range of common divide-and-conquer algorithms, and have shown that most have the parallelism necessary for successful use of the team-parallel model, and that a significant number are unbalanced algorithms that are hard to implement efficiently in other models.

8.1.2   The language and system

I have designed and implemented the Machiavelli system, which is a portable implementation of the team parallel model for distributed-memory message-passing computers. Machiavelli is built as an extension to C, and uses vectors as its collection-oriented data type. There are several basic parallel operation that can be applied to vectors, including scans, reductions, and permutations. There are also two new forms of syntax: the creation of a data-parallel expression operating on one or more source vectors (modelled on the same facility in NESL [BHS+94]), and parallel recursion (similar to that in PCP [Bro95]).

Machiavelli is implemented using a preprocessor, a library of basic operations, and a small run-time system. It compiles to C and MPI for maximum portability. The choice of C also allows the user to easily replace the serial version of a function compiled by the preprocessor with efficient serial code. The basic operations are specialized for user-defined types and their corresponding MPI datatypes, enabling vectors of structures to be communicated in a single message. Load balancing is achieved through function shipping, using a centralized decision-making manager to get around MPI's lack of one-sided messages.

8.1.3   The results

I have used the Machiavelli system to prove my thesis that irregular divide-and-conquer algorithms can be efficiently implemented on parallel computers. Specifically, I have benchmarked a variety of code on three different parallel architectures, ranging from loosely coupled (IBM SP2), through an intermediate machine (Cray T3D), to tightly coupled (a shared-memory SGI Power Challenge). Three types of code were benchmarked.

First, the basic Machiavelli primitives were tested. The primitives scale well, but the cost of all-to-all communication in parallel code is 5–10 times that of simple data-parallel operations, as we would expect from their implementation and the machine architectures. Per-processor performance falls for all primitives involving communication if the problem size is not scaled

134

with the machine size, due to the fixed overhead of the MPI primitives used. Finally, it was interesting to note that all the machines exhibited similar performance, both absolutely and in terms of their scaling behavior, in spite of their very different architectures.

Second, four small divide-and-conquer algorithms were tested, including examples that are very irregular and communication-intensive (e.g., quicksort), and others that are balanced and computation-intensive (e.g., a dense matrix multiply). These showed that Machiavelli code is often very similar in style to the equivalent code in a higher-level language such as NESL, although it imposes on the programmer the additional requirements of declaring and freeing vectors. The algorithms also demonstrated reasonable parallel performance, achieving parallel efficiencies of between 22% and 92%, with the poorest performance coming from irregular algorithms running small datasets on loosely-coupled architectures.

Finally, an early version of the Machiavelli toolkit (with libraries but no preprocessor) was used to build a full-size application, namely two-dimensional Delaunay triangulation. Using a combination of a new parallel algorithm by Blelloch et al [BMT96] and efficient single-processor code from the Triangle package [She96c], this is the first parallel implementation of Delaunay triangulation to achieve good performance on both regular and irregular data sets. It is also the first to be widely portable, thanks to its use of MPI, and has the best performance, being approximately three times more efficient than previous parallel codes.

8.2   Future Work

There are many possible directions for future work from this thesis. Some can be considered improvements to the Machiavelli system in particular, while others represent research into team parallelism in general. I will briefly discuss each topic, concentrating first on possible improvements to Machiavelli.

8.2.1   A full compiler

The most obvious improvement to the Machiavelli system would be to replace the simple preprocessor with a full compiler. The use of a parser in place of the current *ad hoc* approach to source processing would give the user more flexibility in how code is expressed, and would also expose more possible optimizations to the compiler. For example, in the case of a vector that is created and then immediately becomes the target of a `reduce_sum` before being discarded, the current preprocessor creates the vector, sums it, and then frees it. This could be replaced by optimal code which performs the sum in the same loop that is creating the elements of the vector, whilst never actually writing them to memory. In the general case, this would need a full dataflow analysis.

135

An additional vector optimization technique is that of "epoch analysis" [Cha91]. Subject to the constraints of a dataflow analysis, this groups vectors and operations on them according to their sizes, which in turn allows for efficient fusion of the loops that create them.

Finally, the Machiavelli language would be easier to use if the burden of freeing vectors was removed from the programmer. The decision of where and when to free memory is particularly important in the team-parallel model, since the amount of memory involved can be very large, and the programmer must consider both the recursive nature of the code and its mapping onto different teams of processors. However, a conservative run-time garbage collector, such as that provided by Boehm [Boe93], is unlikely to be useful because arguments passed to a function on the stack are in scope throughout future recursive calls. This prevents a conservative garbage collector from freeing them, whereas in a divide-and-conquer algorithm we typically want to reclaim their associated memory before proceeeding. Instead, the compiler and runtime system could implement a simple reference-counting system similar to that in the VCODE interpreter [BHS+94], freeing a vector after its last use in a function.

8.2.2 Parameterizing for MPI implementation

Machiavelli has been written to be as portable as possible, and makes no assumptions about the underlying MPI implementation. Where there is a choice, it generally uses the highest-level primitive available (for example, using MPI_Alltoallv for all-to-all communication in place of simple point-to-point messages). However, different MPI implementations choose different performance tradeoffs, and this can result in the optimal choice of primitive being dependent on the platform [Gro96]. Examples include the choice of delivery options for point-to-point messages, whether collection operations can be specialized for contiguous types, and whether it is faster to build up an index type or to accumulate elements in a user-space buffers. A possible performance optimization would therefore be to parameterize Machiavelli for each MPI implementation. This would involve benchmarking each of the implementation choices for every MPI implementation. The programmer would then use a compile-time switch to select the appropriate set of choices for a particular MPI implementation.

8.2.3 Improved load-balancing system

As explained in Chapter 4, the current load-balancing system in Machiavelli is limited by MPI's requirement to match sends and receives, and by the fact that no thread-safe implementation of MPI is widely available. This leads to the "function shipping" model of load balancing. If either of these limitations were removed, two alternative models of load balancing could be used.

136

The first is a work-stealing system, similar to that in Cilk [BJK+95]. In this model, a processor that has completed its computation transparently steals work from a busy processor. The simplest way to achieve this is to arrange for parcels of future work to be stored in a queue on each processor, which all processors are allowed to take work from. In a recursive divide-and-conquer algorithm, the parcels of work would be function invocations. The transparency of the stealing can be implemented using one-sided messages, in which the receiving processor does not have to participate in the message transfer. Alternatively threads can be used, where each processor runs a worker thread and a queue-manager thread. The manager thread is always available to receive messages requesting work, either from the worker thread on the same processor or from a different processor. Available processors can find a processor with work using either a centralized manager, as in the current Machiavelli system, or a randomized probing system.

Second, a perfectly load-balanced system could be used. In this case, the division of data between processors would be allowed to include fractional processors. Thus, the amount of data per processor would be constant, but each processor could be responsible for multiple vectors (or ends of vectors) at the same time. Using threads, this could be implemented with up to three threads per processor, representing the vector shared with processors to the left, vectors solely on this processor, and the vector shared with processors to the right. These threads would exist in different teams, corresponding to the vectors they represented. A final "shuffling" stage would be necessary, as happens in concatenated parallelism [AGR96], to move vectors entirely to single processors so that they could be processed using serial code, but no centralized manager would be needed.

The recent MPI-2 standard holds out hope that these systems could be portably implemented in future, since it offers both one-sided communication and guarantees of thread safety. However, no implementations of MPI-2 have yet appeared.

8.2.4 Alternative implementation methods

The current version of Machiavelli has been designed with distributed-memory message-passing machines in mind. However, as we saw in the results chapters, it also performs well on small-scale shared-memory machines. If only shared-memory machines were to be supported, changes can be made both in the implementation of Machiavelli and in the programming model of team parallelism.

For Machiavelli, a new implementation layer based on direct access to shared memory could replace the MPI operations, possibly saving both time and space. For example, the vector send operation could be implemented with a single loop on each processor that loads a data element from shared memory, loads the global index to permute it to, maps the global index to an address in shared memory, and writes the data. In the current implementation three

137

loops are used, slowing down the process; one to count the number of elements to be sent to each processor, another to pack the elements into MPI communication buffers of the correct size, and a third to unpack the buffers on the receiving end.

Memory is still likely to be physically distributed on scalable shared-memory machines, and so the same goals of minimizing communication and maximizing locality apply as for message-passing systems. Thus, the team-parallel approach of mapping divide-and-conquer recursion onto smaller and smaller subsets of processors is still likely to offer performance advantages. Additionally, shared-memory systems provide another memory access model, namely globally-shared memory. This allows any processor to access a vector in any other team, in contrast to the team-shared memory that is assumed by the team-parallel model. With appropriate synchronization, globally-shared memory corresponds to the shared H-PRAM model of Heywood and Ranka [HR92a], while the team-shared memory corresponds to their private H-PRAM (see also Chapter 2). Globally-shared memory of this type would enable the implementation of impure divide-and-conquer algorithms, where sibling teams can communicate with each other.

Other transport mechanisms could also be used in place of MPI, although this would decrease the portability of the system. For example, active messages [vECGS92] would provide the one-sided communication that MPI lacks. Another implementation alternative would be bulk synchronous parallelism [Val90], although this would have to be extended to support the concept of separate and subdividable teams that run asynchronously with respect to each other.

8.2.5   Coping with non-uniform machines

The current Machiavelli system and team-parallel model assume that machines are homogenous with respect to both the nodes and the communication network between them. However, the model could easily be adapted for heterogeneity in either respect.

Processor nodes may be heterogenous in architecture, speed, and memory. Architectural differences can impact performance if data types must be converted when communicating between processors (for example, switching the endianness of integers). Thus we would want to bias the algorithm that maps processors to teams so that it tends to produce teams containing processors with the same architecture, in order to minimize these communication costs. Processor speed and memory size differences both affect performance, in that we would ideally like to assign sufficient data to each processor so that they all finish at the same time. This can involve a time/space tradeoff if there is a slow processor with lots of memory, in which case the user must choose to either use all of its memory (and slow down the rest of the computation), or just a subset (and only solve a smaller problem). Note that placing different amounts of data on each processor would break the basic assumption of Machiavelli that vectors are normally balanced, and would require extensions to the basic primitives to handle the unbalanced case.

138

If the interprocessor network is heterogenous in terms of its latency and/or bandwidth (for example, a fat-tree or a cluster of SMP nodes), we again want to bias the selection algorithm. This time it should tend to place all the processors in a team within the same high-bandwidth section of network. Once again, there is a tradeoff involved, since the optimal team sizes at a particular level of recursion might not match the optimal selection of processors for teams. Programming models for machines with heterogenous networks are just starting to be explored. For example, the SIMPLE system by Bader and Jájá [BJ97] implements a subset of MPI primitives for a cluster of SMP nodes that use message-passing between the nodes.

8.2.6 Input/output

As noted in the introduction, I have made no attempt in this dissertation to consider I/O in the context of team parallelism. However, there are some clear directions to follow. For example, all algorithms begin and end with all of the processors in the same team and the data shared equally between them. In this case storage and retrieval of vectors to and from disk is trivial if one disk is associated with each processor. If instead there are specialized I/O processors the vectors must be transmitted across the network, but this is also relatively easy since all the processors are available and each has the same amount of data.

The need for I/O at intermediate levels of the algorithm, where processors are in small teams, is less clear. Some algorithms may be simplified if output data can be written to local disk directly from the serial code. Additionally, check-pointing of long-running algorithms may occur in the middle of team phases, where processors are synchronized within a team but not between teams. This case is also relatively simple to handle if every processor has a local disk, but synchronization becomes much harder if there are dedicated I/O resources (especially if these are also compute nodes, as on the Intel Paragon [Int91]).

A particular problem with I/O at intermediate levels in any load-balanced system is that the nondeterministic nature of load balancing makes the mapping of an unbalanced algorithm onto processors impossible to predict. Even when input data is identical, the asynchrony of the system (due, for example, to interrupts occurring at different times on different processors) can result in race conditions between processors. Thus, although the output is guaranteed, details such as which processor performed which computation are not, and when storing intermediate or final results of an algorithm on disk there is no guarantee that a vector mapped a certain way and stored to disk on one run will correspond to an identical vector on another run.

8.2.7 Combining with existing models

A final possible topic for research is to combine team parallelism into existing programming models and languages. This could be tackled via either translation or integration.

139

In translation, Machiavelli or a similar language would be used as the back end of the compiler for a higher-level language such as NESL. This has the advantage of minimizing language changes visible to the programmer. However, it would require the programmer to choose which back end to use, depending on the properties of their algorithm. In addition, retrofitting a new back end to an existing language generally results in an imperfect fit; there are often fewer opportunities for optimization than in a system written from scratch.

In integration, a language would support both team parallelism and one or more other parallel models. Concatenated parallelism [AGR96] is an obvious choice, since it is in some sense a competitor to team parallelism, and a language that supported both would allow for fairer comparisons. Although this involves considerably more work, it is likely that the different models could share some common code (for example, the serial code to run on one processor would be the same). In the case of concatenated parallelism this would also be the first compiler to support the model, since all concatenated parallel algorithms implemented so far have been hand-written. Alternatively, a language that supported both team parallelism and flattening parallelism, compiling down to a library that combined the Machiavelli runtime and primitives similar to those in CVL [BCH+93], would allow a much greater range of algorithms to be written, since it would not be restricted to expressing only divide-and-conquer algorithms.

8.3 Summary

To summarize, in this dissertation I have shown that:

- There is parallelism available in many useful irregular divide-and-conquer algorithms.

- The team-parallel model can exploit this parallelism with a natural mapping of the recursive nature of the algorithms onto subteams of processors.

- A portable implementation of team parallelism for distributed-memory parallel computers can achieve good empirical performance when compared to serial algorithms and the best previous parallel work.

Bibliography

[ABM94]   Liane Acker, Robert Browning, and Daniel P. Miranker. On parallel divide-and-conquer. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems*, pages 336–343, 1994.

[ACD+97]   K. T. P. Au, M. M. T. Chakravarty, J. Darlington, Y. Guo, S. Jähnichen, M. Khler, G. Keller, W. Pfannenstiel, and M. Simons. Enlarging the scope of vector-based computations: Extending Fortran 90 by nested data parallelism. In *Proceedings of the International Conference on Advances in Parallel and Distributed Computing*. IEEE, March 1997.

[ACG+88]   A. Aggarwal, B. Chazelle, L. Guibas, C. Ó Dúnlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(3):293–327, 1988.

[Ada93]   Don Adams. *Cray T3D System Architecture Overview Manual*. Cray Research, Inc., September 1993.

[AGR96]   Srinivas Aluru, Sanjay Goil, and Sanjay Ranka. Concatenated parallelism: A technique for efficient parallel divide and conquer. Technical Report SCCS-759, NPAC, 1996.

[AHU74]   Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AMM+95]   T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir. SP2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[AS95]   Klaus Achatz and Wolfram Schulte. Architecture independent massive parallelization of divide-and-conquer algorithms. In *Proceedings of the 3rd International Conference on the Mathematics of Program Construction*. Springer-Verlag, July 1995.

[Aur91]   Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, September 1991.

[Axf90]   Tom Axford. An elementary language construct for parallel programming. *ACM SIGPLAN Notices*, 25(7):72–80, July 1990.

[Axf92]   Tom H. Axford. The divide-and-conquer paradigm as a basis for parallel language design. In *Advances in Parallel Algorithms*, chapter 2. Blackwell, 1992.

[Bat68]   Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 307–314, April 1968.

[BC90]   Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings of the Symposium on the Frontiers of Massively Parallel Computation*, pages 471–480. IEEE, October 1990.

[BCH⁺93]   Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Margaret Reid-Miller, Jay Sipelstein, and Marco Zagha. CVL: A C vector library. Technical Report CMU-CS-93-114, School of Computer Science, Carnegie Mellon University, February 1993.

[BDH96]   C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, December 1996.

[BG81]   J-D. Boissonat and F. Germain. A new approach to the problem of acquiring randomly oriented workpieces out of a bin. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, volume 2, pages 796–802, 1981.

[BG96]   Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, May 1996.

[BH86]   J. E. Barnes and P. Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324(4):446–449, December 1986.

[BH93]   Guy E. Blelloch and Jonathan C. Hardwick. Class notes: Programming parallel algorithms. Technical Report CMU-CS-93-115, School of Computer Science, Carnegie Mellon University, February 1993.

[BHLS⁺94] Christian Bischof, Steven Huss-Lederman, Xiaobai Sun, Anna Tsao, and Thomas Turnbull. Parallel performance of a symmetric eigensolver based on the invariant subspace decomposition approach. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pages 32–39. IEEE, 1994.

[BHMT]   Guy E. Blelloch, Jonathan C. Hardwick, Gary L. Miller, and Dafna Talmor. Design and implementation of a practical parallel Delaunay algorithm. Submitted for journal publication.

[BHS⁺94]   Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[BHSZ95]   Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. NESL user's manual (for NESL version 3.1). Technical Report CMU-CS-95-169, School of Computer Science, Carnegie Mellon University, July 1995.

[BJ97]   David A. Bader and Joseph Jájá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). Preliminary version, May 1997.

[BJK⁺95]   Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, Andrew Shaw, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, July 1995.

[Ble87]   Guy E. Blelloch. Scans as primitive parallel operations. In *Proceedings of the 16th International Conference on Parallel Processing*, pages 355–362, August 1987.

[Ble90]   Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[Ble95]   Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon University, September 1995.

[Ble96]   Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[BMT96]   Guy E. Blelloch, Gary L. Miller, and Dafna Talmor. Developing a practical projection-based parallel Delaunay algorithm. In *Proceedings of the 12th Annual Symposium on Computational Geometry*. ACM, May 1996.

[BN94]   Guy E. Blelloch and Girija Narlikar. A comparison of two *n*-body algorithms. In *Proceedings of DIMACS International Algorithm Implementation Challenge*, October 1994.

143

[Boe93]   Hans-J Boehm. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 197–206, 1993.

[BPS93]   Stephen T. Barnard, Alex Pothen, and Horst D. Simon. A spectral algorithm for envelope reduction of sparse matrices. In *Proceedings of Supercomputing '93*, pages 493–502. ACM, November 1993.

[Bro95]   Eugene D. Brooks III. PCP: A paradigm which spans uniprocessor, SMP and MPP architectures. SC'95 poster presentation, June 1995.

[BS90]   Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, February 1990.

[CCC$^+$95] Thomas H. Cormen, Sumit Chawla, Preston Crow, Melissa Hirschl, Roberto Hoyle, Keith D. Kotay, Rolf H. Nelson, Nils Niewejaar, Scott M. Silver, Michael B. Taylor, and Rajiv Wickremesinghe. DartCVL: The Dartmouth C vector library. Technical Report PCS-TR-95-250, Department of Computer Science, Dartmouth College, 1995.

[CCH95]   George Chochia, Murray Cole, and Todd Heywood. Implementing the hierarchical PRAM on the 2D mesh: Analyses and experiments. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing*, pages 587–595, October 1995.

[CCH96]   George Chochia, Murray Cole, and Todd Heywood. Synchronizing arbitrary processor groups in dynamically partitioned 2-D meshes. Technical Report ECS-CSG-25-96, Dept. of Computer Science, Edinburgh University, July 1996.

[CCS97]   L. Paul Chew, Nikos Chrisochoides, and Florian Sukup. Parallel constrained Delaunay meshing. In *Proceedings of the Joint ASME/ASCE/SES Summer Meeting Special Symposium on Trends in Unstructured Mesh Generation*, June 1997. To appear.

[CDY95]   Soumen Chakrabarti, James Demmel, and Katherine Yelick. Modeling the benefits of mixed data and task parallelism. In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1995.

[CGD90]   Richard Cole, Michael T. Goodrich, and Colm Ó Dúnlaing. Merging free trees in parallel for efficient Voronoi diagram construction. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, pages 32–45, July 1990.

144

[Cha91] Siddhartha Chatterjee. *Compiling Data-Parallel Programs for Efficient Execution on Shared-Memory Multiprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon University, October 1991.

[Cha93] Siddhartha Chatterjee. Compiling data-parallel programs for efficient execution on shared-memory multiprocessors. *ACM Transactions on Programming Languages and Systems*, 15(3):400–462, July 1993.

[CHM$^+$90] Shigeru Chiba, Hiroki Honda, Hiroaki Maezawa, Taketo Tsukioka, Michimasa Uematsu, Yasuyuki Yoshida, and Kaoru Maeda. Divide and conquer in parallel processing. In *Proceedings of the 3rd Transputer/Occam International Conference*, pages 279–293. IOS, Amsterdam, April 1990.

[CKP$^+$93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, May 1993.

[CLM$^+$93] P. Cignoni, D. Laforenza, C. Montani, R. Perego, and R. Scopigno. Evaluation of parallelization strategies for an incremental Delaunay triangulator in E^3. Technical Report C93-17, Consiglio Nazionale delle Ricerche, November 1993.

[CM91] B. Carpentieri and G. Mou. Compile-time transformations and optimization of parallel divide-and-conquer algorithms. *ACM SIGPLAN Notices*, 26(10):19–28, October 1991.

[CMPS93] P. Cignoni, C. Montani, R. Perego, and R. Scopigno. Parallel 3D Delaunay triangulation. In *Proceedings of the Computer Graphics Forum (Eurographics '93)*, pages 129–142, 1993.

[Col89] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, 1989.

[Cox88] C. L. Cox. Implementation of a divide and conquer cyclic reduction algorithm on the FPS T-20 hypercube. In *Proccedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 1532–1538, 1988.

[CSS95] Manuel M. T. Chakravarty, Friedrich Wilhelm Schroer, and Martin Simons. V—nested parallelism in C. In *Proceedings of Workshop on Massively Parallel Programming Models*, October 1995.

[CSY95] Timothy M. Y. Chan, Jack Snoeyink, and Chee-Keng Yap. Output-sensitive construction of polytopes in four dimensions and clipped Voronoi diagrams in three.

145

In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 282–291, 1995.

[DD89]   J. R. Davy and P. M. Dew. A note on improving the performance of Delaunay triangulation. In *Proceedings of Computer Graphics International '89*, pages 209–226. Springer-Verlag, 1989.

[DeF87] L. DeFloriani. *Surface Representations based on Triangular Grids*, volume 3 of *The Visual Computer*, pages 27–50. Springer-Verlag, 1987.

[DGT93] J. Darlington, M. Ghanem, and H. W. To. Structured parallel programming. In *Proceedings of the Massively Parallel Programming Models Conference*, September 1993.

[Dwy86] R. A. Dwyer. A simple divide-and-conquer algorithm for constructing Delaunay triangulations in $O(n \log \log n)$ expected time. In *Proceedings of the 2nd Annual Symposium on Computational Geometry*, pages 276–284. ACM, June 1986.

[Erl95] Thomas Erlebach. *APRIL 1.0 User manual*. Technische Universtät München, 1995.

[ES91]   Herbert Edelsbrunner and Weiping Shi. An $O(n \log^2 h)$ time algorithm for the three-dimensional convex hull problem. *SIAM Journal on Computing*, 20:259–277, 1991.

[EW95]   Dean Engelhardt and Andrew Wendelborn. A partitioning-independent paradigm for nested data parallelism. In *Proceedings of International Conference on Parallel Architectures and Compiler Technology*, June 1995.

[FHS93] Rickard E. Faith, Doug L. Hoffman, and David G. Stahl. UnCvl: The University of North Carolina C vector library. Technical Report TR93-063, Computer Science Department, University of North Carolina at Chapel Hill, 1993.

[For92] Steven Fortune. Voronoi diagrams and Delaunay triangulations. In Ding-Zhu Du and Frank Hwang, editors, *Computing in Euclidean Geometry*, pages 193–233. World Scientific, 1992.

[For94] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications and High Performance Computing*, 8(3/4), 1994.

[GA94]   Kevin Gates and Peter Arbenz. Parallel divide and conquer algorithms for the symmetric tridiagonal eigenproblem. Technical Report 222, Department of Computer Science, ETH Zurich, November 1994.

146

[GAR96]   Sanjay Goil, Srinivas Aluru, and Sanjay Ranka. Concatenated parallelism: A technique for efficient parallel divide and conquer. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, pages 488–495, October 1996.

[Ghu97]   Anwar M. Ghuloum. *Compiling Irregular and Recurrent Serial Code for High Performance Computers*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

[GLDS]   William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. http://www.mcs.anl.gov/mpi/mpicharticle/paper.html.

[GLS94]   William Gropp, Ewing Lusk, and Anthony Skjelum. *Using MPI*. MIT Press, 1994.

[Gor96]   S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In *Proceedings of Eighth International Symposium on Programming Languages, Implementations, Logics and Programs*, pages 274–288, 1996.

[GOS94]   T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a High Performance Fortran framework. *IEEE Parallel and Distributed Technology*, 2(3):16–26, Fall 1994.

[GPR+96]   Allen Goldberg, Jan Prins, John Reif, Rik Faith, Zhiyong Li, Peter Mills, Lars Nyland, Dan Palmer, James Riely, and Stephen Westfol. The Proteus system for the development of parallel applications. In M. C. Harrison, editor, *Prototyping Languages and Prototyping Technology*, chapter 5, pages 151–190. Springer-Verlag, 1996.

[Gre94]   John Greiner. A comparison of data-parallel algorithms for connected components. In *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 16–25, June 1994.

[Gro96]   William Gropp. Tuning MPI programs for peak performance. http://www.mcs.anl.gov/mpi/tutorials/perf/, 1996.

[Guh94]   Sumanta Guha. An optimal mesh computer algorithm for constrained Delaunay triangulation. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 102–109. IEEE, April 1994.

[GWI91]   Brent Gorda, Karen Warren, and Eugene D. Brooks III. Programming in PCP. Technical Report UCRL-MA-107029, Lawrence Livermore National Laboratory, 1991.

147

[Har94]   Jonathan C. Hardwick. Porting a vector library: a comparison of MPI, Paris, CMMD and PVM (or, "I'll never have to port CVL again"). Technical Report CMU-CS-94-200, School of Computer Science, Carnegie Mellon University, November 1994.

[Har97]   Jonathan C. Hardwick. Implementation and evaluation of an efficient parallel Delaunay triangulation algorithm. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 1997.

[HL91]   W. L. Hsu and R. C. T. Lee. Efficient parallel divide-and-conquer for a class of interconnection topologies. In *Proceedings of the 2nd International Symposium on Algorithms*, pages 229–240, 1991.

[HMS$^+$97] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. *BSPlib: The BSP Programming Library*, May 1997. http://www.bsp-worldwide.org/.

[Hoa61]   C. A. R. Hoare. Algorithm 63 (partition) and algorithm 65 (find). *Communications of the ACM*, 4(7):321–322, 1961.

[Hoa62]   C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–15, 1962.

[HPF93]   High Performance Fortran Forum. *High Performance Fortran Language Specification*, May 1993.

[HQ91]   Philip J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers*. Scientific and Engineering Computation Series. MIT Press, 1991.

[HR92a]   Todd Heywood and Sanjay Ranka. A practical hierarchical model of parallel computation. I. The model. *Journal of Parallel and Distributed Computing*, 16(3):212–232, November 1992.

[HR92b]   Todd Heywood and Sanjay Ranka. A practical hierarchical model of parallel computation. II. Binary tree and FFT algorithms. *Journal of Parallel and Distributed Computing*, 16(3):233–249, November 1992.

[HS91]   S. Hummel and E. Schonberg. Low-overhead scheduling of nested parallelism. *IBM Journal of Research and Development*, 35(5/6):743–765, September/November 1991.

[HWW97]   Kai Hwang, Cheming Wang, and Cho-Li Wang. Evaluating MPI collective communication on the SP2, T3D, and Paragon multicomputers. In *Proceedings of*

the *Third IEEE Symposium on High-Performance Computer Architecture*. IEEE, February 1997.

[IJ90]   Ilse C. F. Ipsen and Elizabeth R. Jessup. Solving the symmetric tridiagonal eigenvalue problem on the hypercube. *SIAM Journal on Scientific and Statistical Computing*, 11(2), March 1990.

[Int91]   Intel Corp. *Paragon X/PS Product Overview*, March 1991.

[KQ95]   Santhosh Kumaran and Michael J. Quinn. Divide-and-conquer programming on MIMD computers. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 734–741. IEEE, April 1995.

[Mer92]   Marshal L. Merriam. Parallel implementation of an algorithm for Delaunay triangulation. In *Proceedings of Computational Fluid Dynamics*, volume 2, pages 907–912, September 1992.

[Mer96]   Simon C. Merrall. Parallel execution of nested parallel expressions. *Journal of Parallel and Distributed Computing*, 33(2):122–130, March 1996.

[MH88]   Z. G. Mou and P. Hudak. An algebraic model of divide-and-conquer and its parallelism. *Journal of Supercomputing*, 2(3):257–278, November 1988.

[Mis94]   Jayadev Misra. Powerlist: a structure for parallel recursion. In *A Classical Mind: Essays in Honor of C. A. R. Hoare*. Prentice-Hall, January 1994.

[MNP⁺91]   Peter H. Mills, Lars S. Nyland, Jan F. Prins, John H. Reif, and Robert A. Wagner. Prototyping parallel and distributed programs in Proteus. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 10–19, December 1991.

[Mou90]   Zhijing George Mou. A formal model for divide-and-conquer and its parallel realization. Technical Report YALEU/DCS/RR-795, Department of Computer Science, Yale University, May 1990.

[MPHK94]   Kwan-Lui Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4), July 1994.

[MPS⁺95]   Prasenjit Mitra, David Payne, Lance Shuler, Robert van de Geijn, and Jerrell Watts. Fast collective communication libraries, please. In *Proceedings of Intel Supercomputing Users' Group Meeting*, 1995.

149

[MS87]   D. L. McBurney and M. R. Sleep. Transputer-based experiments with the ZAPP architecture. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE: Parallel Architectures and Languages Europe (Volume 1: Parallel Architectures)*, pages 242–259. Springer-Verlag, 1987. Lecture Notes in Computer Science 258.

[MW96]   Ernst W. Mayr and Ralph Werchner. Divide-and-conquer algorithms on the hypercube. *Theoretical Computer Science*, 162(2):283–296, 1996.

[NB97]   Girija J. Narlikar and Guy E. Blelloch. Space-efficient implementation of nested parallelism. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, June 1997.

[Nel81]   Bruce J. Nelson. *Remote Procedure Call*. PhD thesis, Computer Science Department, Carnegie-Mellon University, May 1981.

[Oed92]   Wilfried Oed. Cray Y-MP C90: System features and early benchmark results. *Parallel Computing*, 18(8):947–954, August 1992.

[OvL81]   Mark H. Overmars and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[Pet81]   F. J. Peters. Tree machines and divide-and-conquer algorithms. In *Proceedings of the Conference on Analysing Problem Classes and Programming for Parallel Computing (CONPAR '81)*, pages 25–36. Springer, June 1981.

[Pie94]   P. Pierce. The NX message passing interface. *Parallel Computing*, 20(4):463–480, April 1994.

[PP92]   A. J. Piper and R. W. Prager. A high-level, object-oriented approach to divide-and-conquer. In *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, pages 304–307, 1992.

[PP93]   Jan F. Prins and Daniel W. Palmer. Transforming high-level data-parallel programs into vector operations. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 119–128, May 1993.

[PPCF96]   Daniel W. Palmer, Jans F. Prins, Siddhartha Chatterjee, and Richard E. Faith. Piecewise execution of nested data-parallel programs. *Lecture Notes in Computer Science*, 1033, 1996.

[PPW95] Daniel W. Palmer, Jan F. Prins, and Stephen Westfold. Work-efficient nested data-parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 186–193. IEEE, February 1995.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*. Texts and Monographs in Computer Science. Springer-Verlag, 1985.

[PV81] Franco P. Preparata and Jean Vuillemin. The cube-connected cycles: A versatile network for parallel computing. *Communications of the ACM*, 24(5):300–309, May 1981.

[RM91] Fethi A. Rabhi and Gordon A. Manson. Divide-and-conquer and parallel graph reduction. *Parallel Computing*, 17(2):189–205, June 1991.

[RS89] John H. Reif and Sandeep Sen. Polling: A new randomized sampling technique for computational geometry. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 394–404, 1989.

[Sab88] Gary W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. The MIT Press, 1988.

[SB91] Jay Sipelstein and Guy E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.

[SC95] Thomas J. Sheffler and Siddhartha Chatterjee. An object-oriented approach to nested data parallelism. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*. IEEE, February 1995.

[Sco96] Steven L. Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 26–36. ACM, October 1996.

[SD95] Peter Su and Robert L. Drysdale. A comparison of sequential Delaunay triangulation algorithms. In *Proceedings of the 11th Annual Symposium on Computational Geometry*, pages 61–70. ACM, June 1995.

[SDDS86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[Sed83] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1983.

[SFG+84] Guy L. Steele Jr., Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb. *Common LISP: The Language*. Digital Press, 1984.

[SG97]   T. Stricker and T. Gross. Global address space, non-uniform bandwidth: A memory system performance characterization of parallel systems. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*. IEEE, February 1997.

[SGI94]   Silicon Graphics, Inc. *POWER CHALLENGE Technical Report*, August 1994.

[SGI96]   Silicon Graphics, Inc. *Technical Overview of the Origin Family*, 1996.

[SH97]   Peter Sanders and Thomas Hansch. On the efficient implementation of massively parallel quicksort. In *Proceedings of the 4th International Symposium on Solving Irregularly Structured Problems in Parallel*, June 1997.

[She96a]   Thomas J. Sheffler. The Amelia vector template library. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*, chapter 2. MIT Press, 1996.

[She96b]   Jonathan Richard Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996.

[She96c]   Jonathan Richard Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In *First Workshop on Applied Computational Geometry*, pages 124–133. ACM, May 1996.

[Sip]   Jay Sipelstein. Data representation optimizations for collection-oriented languages. PhD thesis, School of Computer Science, Carnegie Mellon University, to appear.

[SSOG93]   Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 13–22, May 1993.

[Sto87]   Quentin F. Stout. Supporting divide-and-conquer algorithms for image processing. *Journal of Parallel and Distributed Computing*, 4:95–115, 1987.

[Str69]   Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(3):354–356, 1969.

[Su94]   Peter Su. *Efficient parallel algorithms for closest point problems*. PhD thesis, Dartmouth College, 1994. PCS-TR94-238.

[Sun90]   V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[SY97]   Jaspal Subhlok and Bwolen Yang. A new model for integrating nested task and data parallel programming. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, June 1997.

[Too63]   A. Toomre. On the distribution of matter within highly flattened galaxies. *The Astrophysical Journal*, 138:385–392, 1963.

[TPH92]   Walter F. Tichy, Michael Philippsen, and Phil Hatcher. A critique of the programming language C*. *Communications of the ACM*, 35(6):21–24, June 1992.

[TPS+88]   L. J. Toomey, E. C. Plachy, R. G. Scarborough, R. J. Sahulka, J. F. Shaw, and A. W. Shannon. IBM Parallel FORTRAN. *IBM Systems Journal*, 27(4):416–435, November 1988.

[TSBP93]   Y. Ansel Teng, Francis Sullivan, Isabel Beichl, and Enrico Puppo. A data-parallel algorithm for three-dimensional Delaunay triangulation and its implementation. In *Proceedings of Supercomputing '93*, pages 112–121. ACM, November 1993.

[Val90]   Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Eril Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*. IEEE, 1992.

[VWM95]   N. A. Verhoeven, N. P. Weatherill, and K. Morgan. Dynamic load balancing in a 2D parallel Delaunay mesh generator. In *Parallel Computational Fluid Dynamics*, pages 641–648. Elsevier Science Publishers B.V., June 1995.

[Wea92]   N. P. Weatherill. The Delaunay triangulation in computational fluid dynamics. *Computers and Mathematics with Applications*, 24(5/6):129–150, 1992.

[Web94]   Jon Webb. High performance computing in image processing and computer vision. In *Proceedings of the International Conference on Pattern Recognition*, October 1994.

[WM91]   Xiaojing Wang and Z. G. Mou. A divide-and-conquer method of solving tridiagonal systems on hypercube massively parallel computers. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 810–817, December 1991.

[Xue97]   Jingling Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Computing*, 22(12):1621–1645, February 1997.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890